TOWARDS IQ-APPLIANCES: QUALITY-AWARENESS IN INFORMATION VIRTUALIZATION

A Thesis
Presented to
The Academic Faculty

by

Radhika Niranjan Mysore

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2007

TOWARDS IQ-APPLIANCES: QUALITY-AWARENESS IN INFORMATION VIRTUALIZATION

Approved by:

Professor Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Dr. Ada Gavrilovska
College of Computing
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Bonnie Heck Ferri
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: May 2007

ACKNOWLEDGEMENTS

This work would not have been possible without the guidance, support and encouragement of many people. I would like to express my gratitude to them here.

First and foremost, I would like to thank Professor Karsten Schwan and Dr. Ada Gavrilovska for building my graduate career at Georgia Tech. They have not only guided me academically and provided a strong foundation for my work, but also influenced and nurtured me personally. I am indebted to them for making graduate school such a positive experience and will always admire them for their exemplary personalities. I am grateful to Professor Bonnie Heck Ferri and Professor Sudhakar Yalamanchili for their time, invaluable inputs and encouragement towards honing this work. I will always remain thankful for the personal interest they have taken in my academic career.

I have been fortunate to have had an opportunity to learn and interact with extremely talented members of the Kernel Group at the Center For Experimental Research in Computer Systems, Georgia Tech. I must thank Sanjay Kumar not only for helping me with technical issues on more than one occasion but also for being a great friend and making grad-school a pleasant journey. I am also thankful to Himanshu Raj for working with me and helping me out readily all the times I was faced with roadblocks during the integration work. Thanks to Priyanka Tembey for the enjoyable 'chai sessions' and technical additions to this work.

Last but not the least, I must thank my family - my parents, my fiance, and my brother for all the love, support and encouragement they have showered on me. The dreams they see - for me, and with me, inspire me in my goals.

This work is dedicated to all of these wonderful people.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Our research addresses two important problems that arise in modern large-scale distributed systems:

1. The necessity to virtualize their data flows by applying actions such as filtering, format translation, coalescing or splitting, etc.

2. The desire to separate such actions from application level logic, to make it easier for future service-oriented codes to inter-operate in diverse and dynamic environments.

This research considers the runtimes of the 'information appliances' used for these purposes, particularly with respect to their ability to provide diverse levels of Quality of Service (QoS) in lieu of dynamic application behaviors and the consequent changes in the resource needs of their data flows. Our specific contribution is the enrichment of these runtimes with methods for QoS-awareness, thereby giving them the ability to deliver desired levels of QoS even under sudden requirement changes – IQ-appliances. For experimental evaluation, we enrich a prototype implementation of an IQ-appliance, based on the Intel IXP network processor, with the additional functionality needed to guarantee QoS constraints for diverse data streams. Measurements demonstrate the feasibility and utility of the approach. Further, we enhance the Self-Virtualized Network Interface developed in previous work from our group with QoS awareness and demonstrate the importance of such functionality in end-to-end virtualized infrastructures.

CHAPTER I

INTRODUCTION

Modern day distributed infrastructures are increasingly shifting to using hardware or software '*appliances*' to manipulate data flows. These appliances, also known as 'single-function servers' are designed with well defined tasks in mind - be it XML or database accelerators, gaming engines or security devices. Focused towards efficient operation, they also provide ease of integration in large scale distributed systems with simple interfaces.

This trend towards employing appliances is fueled by two other important contemporary developments:

1. Multicore architecture has made it easier to envision one or more cores to be used for information appliances alone to carry out standard functions parallel to other processing.

2. Due to virtualization, applications are now separated from hardware by 2 layers: (i) a resource management layer (virtual machine monitor) and (ii) the operating system. Appliances can exploit their network nearness for low latency of application data preprocessing before such data is presented to the application.

When multiple applications / virtual machines are multiplexed over multiple such appliances, it becomes important to provide:

1. quality of service guarantees so as to guarantee system performance to critical applications;

2. efficient utilization of resources for these information flows; and

3. isolation of flows of different applications.

Towards this end our research identifies and designs feasible runtime mechanisms to incorporate quality awareness in future information appliances. These mechanisms must enable dynamic/application-specific reconfiguration of runtime resources, so as to (i) better meet quality requirements on individual flows or classes of flows; and (ii) better utilize platform resources.

1.1 Background

Modern large-scale distributed applications are constantly faced with data interoperability and consequent performance issues. Examples of these are operational information systems (OISs) used by large companies in their daily operations [43, 20] or the 24/7 engines used by online information services (e.g., online ticket pricing and reservation engine [34]). These can be categorized as information intensive applications since they have to handle large amounts of input data, process them at guaranteed rates, and provide output in specified formats. Also included in this spectrum are online high-performance scientific applications, such as distributed real-time collaboration or remote data visualization [60] which have to deal with large data flows.

There is an increasing gap between the data rates of such information-intensive applications and the ability of general purpose platforms to efficiently access and manipulate that data, in addition to performing the many application-specific tasks required of these processors. To deal with this issue, it has been shown useful to associate with such platforms specialized 'information appliances', which are software components running on separable, potentially custom infrastructures that carry out tasks pertaining to data interoperability, tracking data movements and evaluating them, etc., simple examples including firewalls or intrusion detection engines. By separating such actions from the basic business logic executed by applications, it then

becomes possible to dynamically add new data-centric services, to meet the needs of additional data streams or to deal with new types of data, and to dynamically manage how data streams are manipulated.

Entire businesses are based on the 'appliance' model explained above, serving specific application communities, such as the medical form conversions necessary in hospital and insurance settings, or ensuring the efficient and flexible distribution of data in wide area settings [54]. Examples of such appliances range from firewalls and NAT boxes, to appliances for deep-packet processing, such as content-based load balancers, or units which perform select crypto- or XML-related functionality. A custom appliance used by one of our industry partners in the OIS domain extracts incoming data messages and reformats them in the internal binary data representation [43]. Examples in a search engine context such as at Worldspan, are an appliance performing request load balancing across the engine's many cluster servers, or one that merges generated ticket pricing information with customized DoubleClick advertising based on current user profiles [51]. In all such cases, the purpose of these appliances is essentially, to 'virtualize' some original data stream [56], by manipulating the stream and/or creating a new stream with content suitable for some application-specific purpose.

Given the examples described, it is not surprising that many modern enterprise systems contain separate racks of custom boxes, termed 'barnacles' in [44], each of which serves some distinct purpose. Recent industry efforts aim to consolidate this 'appliance' space, for the same power/space/increased utilization arguments that have been driving virtualization technologies for host systems or in the network domain [40], and targeting a limited set of 'deep-packet' processing functions. The main aspect unique to all these examples is that the appliances are 'network-near'. They are able to execute selected data processing actions 'close' to the network itself, thereby not imposing unnecessary data movement overheads and associated memory

loads on the machines that run business logic.

1.2 Motivation

Since information appliances are shared among different applications in enterprise systems or among virtual machines in virtualized infrastructures, providing guarantees to the applications / VMs is becoming more difficult. This issue is not restricted to appliances. In virtualized environments, caches, network devices, I/O devices are shared between multiple virtual machines. Without quality of service, system level hardware virtualization is subject to anarchy, or disruptive interference from other codes or applications running concurrently. In large scale enterprise systems, some application flows may be more important than the others. One way to attain guarantees on delay / bandwidth is to over-provision resources in such environments for latency/ bandwidth critical applications. However this is not always a cost effective / feasible /manageable solution. To avoid over-provisioning, quality of service notions need to be built into the appliances to provide guarantees for applications that have strict latency requirements.

Quality awareness is difficult to implement for several reasons. First, since it is hard, if not impossible, to statically assess the dynamic resource requirements of a certain data flow and the content-based services that operate on it, due to data-dependencies in services, static specifications have to be enhanced with runtime monitoring and assessment techniques [22]. Second, the fixed up-front partitioning of resources is not typically feasible, as it must be based on worst case predictions and will therefore, result in low overall resource utilization. Worse, it may prevent a select application-flow or service to deliver adequate performance under increased demand, although the platform may have an abundance of unused resources. Third, there may be runtime changes in the parameters that most significantly impact the quality levels experienced by a certain application flow, because these may depend on the specific

application-level processing being performed and the associated quality attributes of interest (e.g., throughput vs. response time).

The specific goal and key contribution of the work described in this thesis is to address these issues and design a set of runtime mechanisms which will

1. provide priority notions to data flows based on that applied to the application itself;

2. maintain this priority despite changes in pattern of incoming data flows or other such external influences;

3. adapt quickly to changes in application needs, and new applications / priorities; and

4. adjust monitoring frequencies based on available compute resources and system properties.

We define appliances capable of such quality awareness to be *IQ-appliances*.

IQ-appliances will have several benefits. First, since quality awareness is ingrained in every stage of the pipeline of data flow processing, from the application to the device, we come closer to achieving end-to-end quality assurances. Second, since these appliances have a feedback loop based on current system parameters, system stability and predictability can be ensured. Third, the responsiveness of the IQ-appliance thus designed helps sensitive and critical applications. Automating the reconfiguration of scheduling in these appliances based on system parameters helps ease maintainability and reduces management costs. Finally, reconfigurable monitoring based on system needs, enables reduction in overhead when the system is operating under stress.

All in all, while quality unaware appliances provide ease of integration and efficiency, IQ-appliances would additionally, at small costs, provide the benefit of closer scheduling coupling and responsiveness.

1.3 Thesis-Statement

In summary, it is possible and important to incorporate responsiveness to quality requirements of applications into appliances by adapting dynamic resource management capabilities.

1. While static partitioning of resources is easy to implement, it proves to be counter productive during stress conditions.

2. Adaptive scheduling is not only feasible to implement but also provides for better utilization of resources.

3. Monitoring can be reconfigurable and used to provide need-based system responsiveness.

1.4 Main Contributions

We describe *an architecture for IQ-appliances and their runtimes*, the latter providing the basic support for flexible online monitoring, resource allocation, and adaptation needed by higher level quality management methods. Such runtime functionality makes it easy to:

- dynamically change how distinct data flows are mapped to platform resources, the goal being to adapt these mappings to better meet current flow constraints with available resources;

- drive these changes with application-level quality notions; and

- provide this changes using monitoring and scheduling components which introduce low overheads.

The runtime's API can be used by applications to:

- respond to dynamic changes in an application specific manner; and

- provide the runtime with information useful for further specialization.

As part of this architecture we propose mechanisms that can be used to provide dynamic quality of service guarantees to applications and discuss its capabilities, including methods for continuous monitoring and configuration state management. We argue that these constitute a sufficient basis for runtime methods for data stream adaptation. We describe the realization of these mechanisms in the context of a prototype implementation of an IQ-appliance developed in our research, using the IXP network processor. Measurements of the prototype demonstrate the feasibility and utility of these 'IQ' capabilities of future information appliances. Specific experimental results concern the overheads of effective stream scheduling and the ability to dynamically adjust the appliance-level resources used by certain information streams. We show that the overheads associated with corresponding monitoring and reconfiguration are low. Current results assume that different quality properties are associated with different virtual flows passing through the appliance, but this may be generalized to differentiate different sub-streams or types of data within a single stream, as well. We also demonstrate the feasibility and advantage of QoS awareness in virtualized infrastructures by building QoS capability into a Self Virtualizing NIC [48].

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the basic functionally and runtime support needed quality-aware information appliances – IQ-appliances. Next, we describe an appliance prototype based on the Intel IXP2400 network processing platform in Chapter 3. Chapter 4 presents measurement results and their analysis. A brief summary of related work and concluding remarks constitute the Chapters 5 and 6.

CHAPTER II

IQ-APPLIANCES: HARDWARE AND SOFTWARE ARCHITECTURE

This chapter motivates in greater detail the design of IQ-appliances by describing the various mechanisms that enable their functionality. We first explain the basic concepts and definitions of terms used in this dissertation. In Section 2.2, we bring out the main assumptions regarding hardware which can support such IQ-appliances, drawing from the learnings from existing hardware support for appliances. The high level design of architecture of an IQ appliance, and its main mechanisms that constitute it are elaborated on in Section 2.3. This is followed by further detailing of the specific mechanisms which enable dynamic resource management. Finally, we present a discussion on how this work is relevant to domains beyond appliances, as demonstrated by our modelling of QoS notions into devices which can be built to be logical devices. Such quality aware logical devices can provide guarantees in end-to-end virtualized infrastructures.

2.1 Basic Concepts

We define an *IQ-appliance* as an information appliance that is able to support differentiated quality levels for distinct application-level data flows, where resource requirements may differ across flows and where each such flow may be virtualized via associated processing actions. To realize such flows, the appliance must implement the following abstractions:

- An *IQ-flow* is an abstraction for a collection of packets or application-level *data items* that can be classified according to some set of predicates. For instance,

an IQ-flow may be all network traffic associated with a certain virtual machine and identified via some virtual identifier (e.g., as supported in protocols like SCTP), or it may be a collection of all application-level data items of a certain type and with a common destination address. An example of the latter is the virtualization of a stream of business events prior to sharing them with an entity external to the company, which involves extracting company-sensitive information and reformatting them into some standard format like XML.

- A *data item* in a flow may correspond to a single or some collection of application-level messages, but it must be fully contained within a single flow and within a limited time window.

- Arbitrary processing actions, *IQ-handlers*, may be associated with each IQ-flow and applied to each data item, ranging from simple data forwarding, replication, or filtering actions, to more complex format translation, parsing, or content down-sampling operations. Examples of the latter include compression in media flows [59] and element selection in remote data visualization [60].

- A flow can specify lower bounds for QoS, which requires the IQ-appliance's runtime to provide certain minimum resource levels, independent of momentary spikes in data rates or in the amounts of processing required by other flows. Our current implementation provides such minimum guarantees, but leaves it to operating systems or applications to provide higher level methods for admission control.

We note that typically, there is substantial flexibility in how IQ-appliance resources may be allocated. For instance, runtime resources beyond those required for lower bound guarantees can be shared so as to first address the quality requirements of the highest priority or most critical flows. Experiments with different strategies for prioritization appear in Section 4 below.

2.2 Hardware Assumptions

Before describing the basic architecture of an IQ-appliance and its specific runtime components evaluated in this work, we first digress by describing in more detail some of the assumptions we make about the hardware on which IQ-appliances run.

An IQ-appliance must deal with multiple information flows and for each flow, it must execute application-specified operations, potentially applied to each of the flow's data items. The highly parallel nature of such processing indicates the need for many different processing cores capable of performing a range of processing actions or execute entire chains of transformations, for many concurrent flows. Modern network processors [25] have such capabilities, albeit in many cases, with some limitations in terms of the processing actions they can carry out (e.g., no floating point support, limited state per operation, etc.). We dismiss the use of parallel ASIC platforms, since that would negate the commoditization argument advanced in Chapter 1. Instead, we assume an appliance architecture similar to those present in current multicore machines, where parallel processing cores have access to shared memory, with a memory hierarchy that ranges from faster and smaller on chip memory (e.g., for maintaining per flow state, or runtime configuration parameters), to larger and slower off-chip memory for storing flow data. Concerning processing, it is reasonable to assume that information appliances will have certain accelerators for frequently executed communication functions, such as protocol processing, hash units for operations like classification, or even cryptographic support. IQ-appliances leverage the presence of such accelerators for application-specific transformations applied to data 'in transit' through the appliance [50].

Finally, while the bulk of the platform's computational resources are designated for fast path processing of different application flows, we assume the existence of a designated control context (e.g., a hardware thread, or a dedicated control processor), which has access to all of the platform's resources and can execute management and

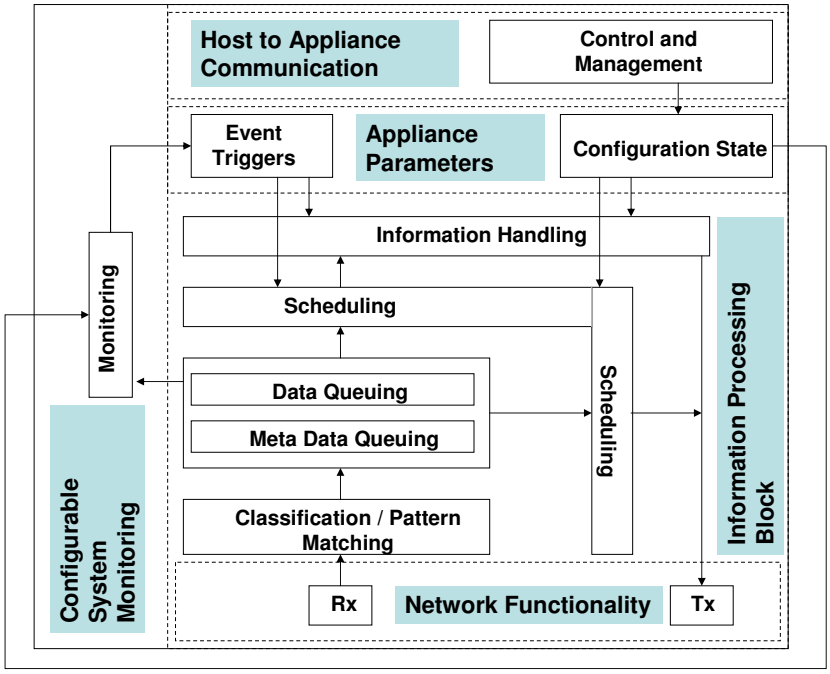

Figure 1: Runtime Components

reconfiguration functionality. This is a reasonable assumption given the now ubiquitous presence of such management processors in bladeservers, for cluster machines, and for local area compute infrastructures [55].

2.3 IQ-Appliances: Runtime Components

Figure 1 illustrates the main runtime components of an IQ-appliance, summarized as follows:

- **Networking Services** – encapsulates tasks related to packet receipt (Rx) and transmission (Tx) and basic protocol-level functionality, such as fragmentation and reassembly.
- **Information Processing Components**

– **Classification / Pattern Matching** – enables flexible matching of incoming packets into flows, where flows may be described by their network-level parameters, such as source and destination IP addresses, virtual flow identifiers, or higher-level information contained into application-level headers, such as data type, or even application-level content.

– **Queuing (Buffering)** – provides functionality to create and manage coordinated access to *buffers of application data* (i.e., packet payload), as well as *metadata queues* (i.e., handles for packet buffers containing header information). These shared memory buffers are the main mechanism for communication among different processing elements on the platform.

– **Information Handling** – in order to enable the per flow execution of arbitrary application-specific functionality and to perform various transformation or filtering/forwarding actions based on flow data, the runtime provides a collection of resources (hardware or software contexts) for the execution of application provided handlers. It is the runtime's responsibility to correctly associate data events from their queues to the appropriate handler. These mappings are maintained in per flow configuration state; they can be dynamically modified.

– **Scheduling** – a set of processing contexts (i.e., threads and/or cores) dedicated to moving processed data from the corresponding data buffers to the shared transmission resources (i.e., threads executing the Tx code and accessing the shared network ports). Similar services can be used to coordinate access to other shared resources, such as crypto units. The exact manner in which the scheduling contexts (or cycles) are allocated to individual flows depends upon the current platform configuration state.

- **Monitoring** – a collection of monitoring "knobs" that can be turned on or off, or whose frequency can be adjusted. These can be used to collect in a predetermined location, information about the current platform state, from individual queue lengths, to per-flow packet delays, etc and the configuration state. Monitoring also generates event triggers after collecting platform state.

- **System Parameters**

  - **Configuration State** – a description of the current values of all of the runtime's configurable parameters, such as the fields used in the classification process, the number of buffer queues, number of cycles or threads dedicated to servicing a particular queue, monitoring behavior, etc.

  - **Event Triggers** – the runtime can generate events associated with different states or changes in operating conditions, such as data rates, queue lengths, etc., with which application-specific response handlers can be associated. For instance, under high load conditions, a low priority flow may choose to apply its custom data handler, rather than letting the runtime drop arbitrary data packets due to buffer overflow.

- **Control & Management** – a separate control entity, which is a software thread or a dedicated control context that is responsible for interaction with external application components and with the management infrastructure and for mapping the externally specified policies or quality requirements into concrete platform configurations. This is done in a manner that is safe with respect to the current runtime operations (i.e., to obtain currently held locks, to prevent inconsistent state of data currently processed, etc.). The actual nature of these mappings is not the focus of this work and will be investigated in elsewhere. Finally, due to resource limitations on the physical appliance platform, the control and management functionality may be partially executed on an *external,*

general-purpose, configuration core.

As illustrated in Figure 1, once classified, data may be immediately enqueued for later processing, or it may immediately be passed for information handling. This implies that the runtime supports *pre-queuing* data handling, for instance, so as to filter out unneeded data before any loads are placed on the platform's memory subsystem. Similarly, we support *post-queuing* data handling, needed for instance to inline certain data manipulations jointly with the transmission-related operations, as in packet and/or header compression to meet throughput levels available on a lower speed link, or to facilitate link fragmentation and interleaving to prevent large packets from causing head of line blocking for delay sensitive data.

The ability to apply in-transit arbitrary application handlers has been shown useful in previous work [19, 18]. Toward this end, we represent application-specific processing codes as lightweight IQ-flow handlers, which can be efficiently applied to data at various points in the data path as it traverses the network. One of the key properties of these handlers is their ability to efficiently interpret, access, and manage the layout of application-level data and the contents of its fields across a set of network packets, i.e., a data unit. A requirement imposed on IQ-flow handlers is that the runtime exposes an API that can provide information about the behavior these application-provided codes, such as instruction counts, need for special purpose hardware, whether they are CPU or I/O bound, etc. Such *IQ-flow handler profiles* can be used to further optimize runtime behavior. We are not concerned with how this information is obtained, but there are multiple options. One option relies on the tool chain used to create handlers. Another option is to have the runtime maintain monitors and counters to extract these characteristics of its various loads. Further discussion of IQ-flow handlers appears in [2].

2.4 Dynamic Platform Resource Management

Next, we describe the set of mechanisms and accompanying data structures needed to permit the dynamic management of platform resources and their use for servicing different IQ-flows. The basis for supporting dynamic resource management lies in the following platform capabilities:

1. Flexible and configurable monitoring of platform parameters.
2. Rich configuration state regarding a range of runtime properties.
3. The ability to perform dynamic platform resource allocation.

The following key mechanisms enable the above mentioned capabilities in IQ-Appliances.

2.4.1 Resource Management Points

Our approach to enabling dynamic platform resource allocation is to enrich the datapath with Resource Management (RM) points. These are execution points in the fast path where it is safe to perform reconfiguration operations, e.g., at data item boundaries. At RM points, individual threads/cores participating in the fast path data processing inspect the configuration state (on a per thread basis) to determine the set of actions in which they should be involved. Examples include

1. determining monitoring frequency and the platform parameters to be monitored;
2. the IQ-handlers to be invoked for particular data items; and
3. the data queue to be serviced based on QoS agreements and scheduling policy, etc.

2.4.2 Configuration State

Platform-resident configuration state is used for multiple purposes. First, it can represent individual flows and the processing functionality associated with flow data, i.e., classification of incoming data messages into appropriate platform queues, and invocation of corresponding IQ-handlers. Second, it is used to allocate platform resources to individual flows. Specifically, at RM points, each processing context (i.e., thread) inspects its configuration state to determine which flow (i.e., queue) or set of flows it should service next. In order to minimize the overheads encountered during this operation, we assume that in the case of multiple flows, these are already ordered according to their priority by the external configuration core. Finally, portions of the configuration state describe the platform monitoring functionality – the concrete set of platform parameters to be monitored, and the frequency with which their values are to be inspected.

This state is organized as global, per-flow, and per-thread tables. Access to the appropriate state components is enabled either by static offsets (for static global or per-thread data structures), or via a flexible and configurable classification process. Such a classification mechanism is necessary to map variable length flow descriptors that may span network-headers (i.e., source and destination IP addresses) or application-headers (i.e., binary format descriptors) to a fixed, smaller-bit size unique identifier. The ability to provide flexible classification like this lies in the selection of parameters to the hash function polynomial, the details of which are not the focus of this work.

Clearly, efficient access to various components of this state is of critical importance to platform performance. Due to its potential size, it is important to realize that it may not be feasible to maintain all of this state in fast, on-chip memory. Instead, we rely on distributing state components along the memory hierarchy on the target hardware platforms (see Section 2.2) and dynamically placing state components in

appropriate memories. A specific example are the large classification tables all of which cannot reside in fast memory at all times. To accommodate this fact, we extend the platform-level flow identifier with location attributes that identify current table location.

2.4.3 Configurable Monitoring

Performance information, either regarding the appliance platform overall, or regarding individual per-flow service levels, can be extracted from a range of runtime parameters, such as core CPU utilization, device Rx/Tx rates, or queue lengths. The exact values to be monitored are described in a subset of the configuration state, along with the frequency of this operation. Monitoring is a distributed mechanism, in that each core monitors the above mentioned values separately, and updates the corresponding resource state entries accessible from the control core.

2.4.4 Consistent Operation

Changes in configuration state may be driven by external action – pushed onto the appliance platform through the external configuration core, or as a result of runtime changes. The control core is involved in pushing the state updates to their appropriate locations in the configuration state, and through the use of efficient bitwise locking scheme, to ensure consistency with respect to fast path flow processing.

2.4.5 Further Optimizations

Our prior work has demonstrated the feasibility and utility of dynamic code swapping [32]. While in the current approach, the runtime actions are interpreted by accessing corresponding state elements, we acknowledge the utility of dynamically hotswapping new instruction store contents, and executing customized generated code. This results in additional performance benefits due to elimination of repeated memory accesses.

2.5 Beyond Appliances

The mechanisms proposed so far are not restricted to appliances and can be applied in various other areas which have to deal with application dynamics, resource allocation and notions of priority. This section focuses on a few such areas where quality of service notions become important. We then go on to elaborate the high level design for incorporating the suggested mechanisms into self-virtualizing device as an example for one such area. Along with information handling capabilities, this device can provide QoS guarantees to VMs and applications and hence behave as a enhanced virtual device referred to as a logical device, with additional attributes/ functionality not natively supported by corresponding physical device [46].

2.5.1 Multicore Platforms

Multicore architecture makes way for performance speedup via parallelization. Current research trends are moving towards maximising application performance in multicore architectures via managed runtime environments [65]. Due the dynamic nature of applications multiplexed on several cores, corresponding cache, memory hierarchy and attached I/O devices, quality of service notions in resource allocation policies of the runtime become important not only to provide guarantees but also to improve utilization. This issue has been addressed by different advances in research and technology. For example in [61], the authors discuss the need for data-stream quality of service for all active cores on a Chip Multi Processor (CMP). On the same lines, [28, 29] also reject cache partitioning schemes and address incorporation of quality of service in allocation policies for cache. ARM's MPCore architecture incorporates the advanced extensible interface (AXI) to enable programming for 'tight' QoS guarantees. Thus with innovation in hardware there is a growing need to address quality of service awareness in software managing such hardware to provide better utilization.

In this work we identify the key mechanisms which provide the capability of reconfigurable resource allocation and monitoring. The notions of configuration state, resource management points and reconfigurable monitoring can be applied in the managed runtime environments of multicore platforms to provide the much needed notions of quality awareness and coupling of application needs with scheduling of device and resource accesses.

2.5.2 Network Processors

The need for Quality of Service notion in networking devices on end systems, particularly in Data center environments has been present for quite sometime now. Cavium Networks [12] has introduced a multi-core network processor to address network-service applications and claims to include a peripheral block solely for the purpose of providing QoS. Similarly Intel IXPs [25] also provide sufficient hardware support which makes it easy to provide for a layer of software pertaining to QoS notions. With such advances in network hardware, it is feasible to support runtime mechanisms described above to provide guaranteed service to network flows. The on board memory on the NIC can be used to house the configuration state and the processor threads can make scheduling decisions based on this state. The configuration state on the NIC can be customized to reflect the current scheduling priorities on the host processor, to provide guaranteed throughput / latency to the corresponding flows. Such QoS aware NICs employed in data centers move us one step closer to the notion of a 'Dark Data Center' [14] - which is a network environment and infrastructure that can be managed remotely and is completely automated.

In order to demonstrate the feasibility and utility of incorporating QoS awareness into NICs we have integrated the runtime mechanisms described in Section 2.3 into a Self Virtualized NIC (S-VNIC) developed as part of the work in [48]. With information processing capabilities included the S-VNIC can now be considered a logical

device capable of providing services to the applications in a virtual machine. We elaborate on this in the next subsection.

2.5.3 QoS Aware Logical Devices

We define logical devices to be similar to appliances, in that they are capable of processing data flows. However they differ in that it is possible to offload a set of application semantics onto the device, and the device is capable of applying corresponding handlers to the data flows - whereas an appliance is generally designed with a particular functionality in mind. [32, 19] developed the notion of logical devices. [48] enabled self virtualization support for such devices - but as is common to other virtualization infrastructures, partitioned resources between different data flows from VMs (virtual flows). The virtual flows are classified into different buffer queues and the logical device services the queues in a round robin fashion. Just as self-virtualizing capability is built into the device, quality awareness can be incorporated. Such a device provided with information handling capability can be used as a logical device.

Following are the main design decisions taken in order to incorporate quality awareness in a self-virtualized device attached to a virtualized host which can be used as a logical device:

1. Configuration State - a table mapping virtual flows to priority is maintained. This table is pre-sorted in the order of priority to reduce the compute cycles required for monitoring. This ordered state is assumed to be downloaded from the host into the NP. Apart from this information, meta data regarding the head and tail pointers to the queues corresponding to the virtual flows are maintained. This data is useful in determining queue lengths which in turn influence scheduling decisions. The state also determines the frequency of monitoring.

2. Scheduling - As long as the higher priority queues do not build up beyond a

specified limit (application specified), an equal number of processing contexts is allocated to all virtual flows and round robin scheduling is used. Once the higher priority queues start building up, priority based scheduling kicks in, with a larger share of the processing contexts allocated to higher priority flows, and a pre-decided number of contexts servicing the lower priority flows to prevent starvation.

3. Monitoring knobs - These are turned on as per host directive and are as frequent as recommended by the host. As before, packet delays, queue lengths and priorities of virtual flows are monitored from time to time by reading configuration state and flow meta data. Monitoring can be distributed or centralized, in the sense that each processing context can separately monitor parameters important to it, or a central thread can be used for that purpose.

4. Event Triggers - Under high load conditions, we choose to change scheduling policies. Data handlers for lower priority flows can also be chosen based on such triggers.

2.6 Chapter Summary

In this section we defined terms we will use in the rest of the dissertation. We also described the runtime mechanisms required for providing quality awareness in information appliances and elaborated how these could be defined to provide dynamic resource management capabilities. Finally we discussed the generality of such mechanisms and presented a few examples. In the next chapter we will discuss the implementation of the high-level design elements we described in this section. We also elaborate on the implementation of quality awareness in the S-VNIC.

CHAPTER III

IMPLEMENTATION DETAIL

We next describe in more detail a concrete realization of an QoS-aware information appliance. For this we first set the ground by describing the Intel IXP2xxx network processors which is our chosen platform for prototype implementation. In Section 3.2 we describe in detail implementation of QoS awareness in information appliances. Finally we describe the implementation of a QoS awareness in a Self-Virtualized NIC which can be used as a logical device in Section 3.3 followed by the chapter summary.

3.1 Intel IXP2xxx Network Processors

Our prototype implementation is based on network processors from the Intel IXP family, specifically the IXP2400, attached to standard Linux-based hosts, to represent these future platforms. As with other such platforms, the key attribute of the Intel 2400 is the presence of multiple on-board processing engines, termed microengines, which can be organized in arbitrary manner to perform certain tasks. Important features of the IXP2400 architecture include:

1. Specialized hardware to support network operations, which gives the ability to implement network services with high packet throughput and low latency [25]. In addition to network centric operations, it also provides support for data processing in the form of CRC checksum calculations, integer arithmetic etc.

2. Microengines - eight 32-bit programmable engines specialized for network processing; these MEs handle the main data plane processing per packet.

3. Eight threads per ME can run with no overhead for context switching.

4. A Linux-based 32bit RISC Xscale core is used for management and other functions.

5. A rich memory hierarchy which includes:

   (a) a fast on-chip 640 32 bit local memory for each microengine;

   (b) global scratchpad memory - a 16-Kbyte storage for general-purpose use with atomic operations and ring support;

   (c) SRAM Controller; and

   (d) DRAM Controller;

   The last 2 provide access to the off chip SRAM and DRAM memory.

6. An integral PCI interface with three DMA channels.

7. The Media and Switch Fabric Interface (MSF) which is an interface for network framers and/or Switch Fabric. It contains receive and transmit buffers.

8. A SHaC unit which contains three main subblocks:

   (a) Scratchpad Memory as previously described in memory hierarchy components.

   (b) Hash Unit which can be used by the Intel XScale core and Microengines to offload hash calculations.

   (c) CAP(CSR Access Proxy) - Chip-wide control and status registers which provide special interprocessor communication features to allow flexible and efficient inter-ME and ME-to-Intel XScale core communications.

9. Performance Monitor counters that can be programmed to count selected internal chip hardware events. They are used to analyze and tune performance.

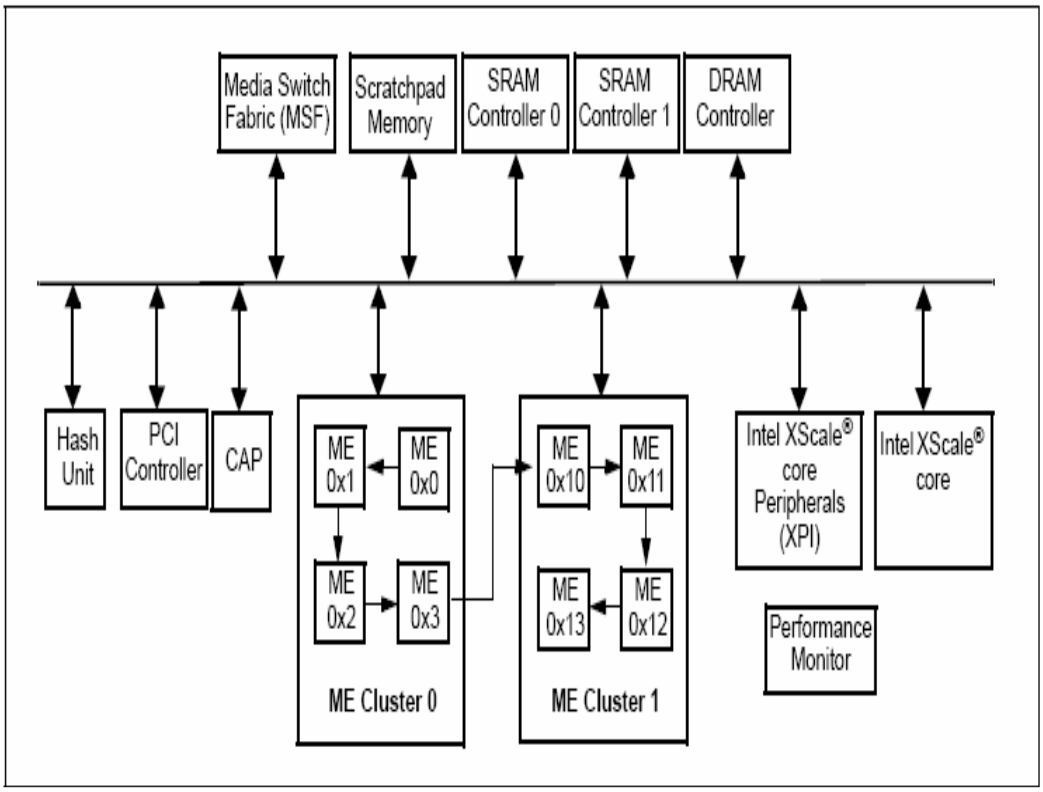

Figure 2: IXP 2400 Network Processor [27]

Figure 2 shows the functional units of the IXP2400 network processor. For more detailed information on the IXP2400, we refer the reader to [26]. The Radisys ENP2611 board [45] on which the IXP2400 resides includes a 600MHz IXP2400, 256MB DRAM, 8MB SRAM, a POS-PHY Level 3 FPGA which connects to 3 Gigabit interfaces and a PCI interface. The Xscale core, running Linux, is primarily used for initialization, management and debugging. The IXP2400 is attached to hosts running standard Linux kernels over a PCI interface. Data is delivered to and from the host-resident application components through the IXPs network interfaces. There is wide range of development tools available which aid the application development for IXP Programmers. These include workbench simulators, SDKs, debugging utilities (data watch, memory watch etc), performance-gathering utilities, etc.

3.2 IXP2400 as a QoS-Aware Information Appliance

The realization of the appliance prototype and the accompanying resource management mechanisms described in Chapter 2 can be summarized as follows.

The processing cores, i.e., the execution of fast path tasks like data receipt and transmission, and the application-provided IQ-handlers, are mapped to threads on the IXP's microengines. The IXP appliance is attached to an x86 host, which serves as the external configuration core. The control core functionality is executed by threads on a single dedicated microengine, with an XScale-resident component involved in signaling and communication with the external processor. RM points are embedded into fast path operations, and they are invoked with configurable frequencies, i.e., they can be invoked say after processing each n application-level data items or for every data item. Monitoring operations are also executed at RM points, potentially with different granularity, and currently focused on monitoring various data rate and queue length indicators.

An IQ-flow is identified as a combination of networking information, i.e., source

address, and application-level information comprised of an efficient binary data type descriptor [10]. The classification process maps this into a configurable smaller bit-size identifier and classifies incoming data into DRAM-resident data buffers. Metadata queues are maintained in faster SRAM memory, with queue descriptors (including tail and head pointer, QoS-related information, etc.) residing in on-chip Scratchpad memory. The configuration state is distributed across microengines' local memories, Scratchpad and SRAM.

Current resource management operations focus on allocating microengine threads to service distinct IQ-flows. Such thread scheduling is described in per thread information, updated by the control core. It can result in several thread allocation schedules, further described and evaluated in Section 4. This work and some of the results also appear in [41, 42, 17].

Mechanisms necessary for Quality Awareness viz. monitoring, scheduling and RM points can also be included at different stages in the pipeline shown in Figure 3 - Rx, packet Processing and/or Tx.

However configuration state is shared across the pipeline.

3.3 IXP2400 as a QoS-Aware Self-Virtualized NIC

We have also enhanced the self-virtualized NIC (S-VNIC) prototype [48] with priority-based QoS support, where flows from different VMs can be assigned different priorities. The NIC as can be seen in Figure 4 provides virtual interfaces directly to the guest VMs, with minimal Virtual Machine Monitor (VMM) interaction in the network I/O path. A device driver for the virtual NIC interface forms the VM-side endpoint. The device driver provides an IOCTL-based interface to the guest VM for communicating QoS requirements, i.e., a numeric priority value. The path responsible for S-VNIC management can be modified to incorporate the communication of the QoS attribute, which can be implemented as a VM-VMM *hypercall*. For proof-of-concept,

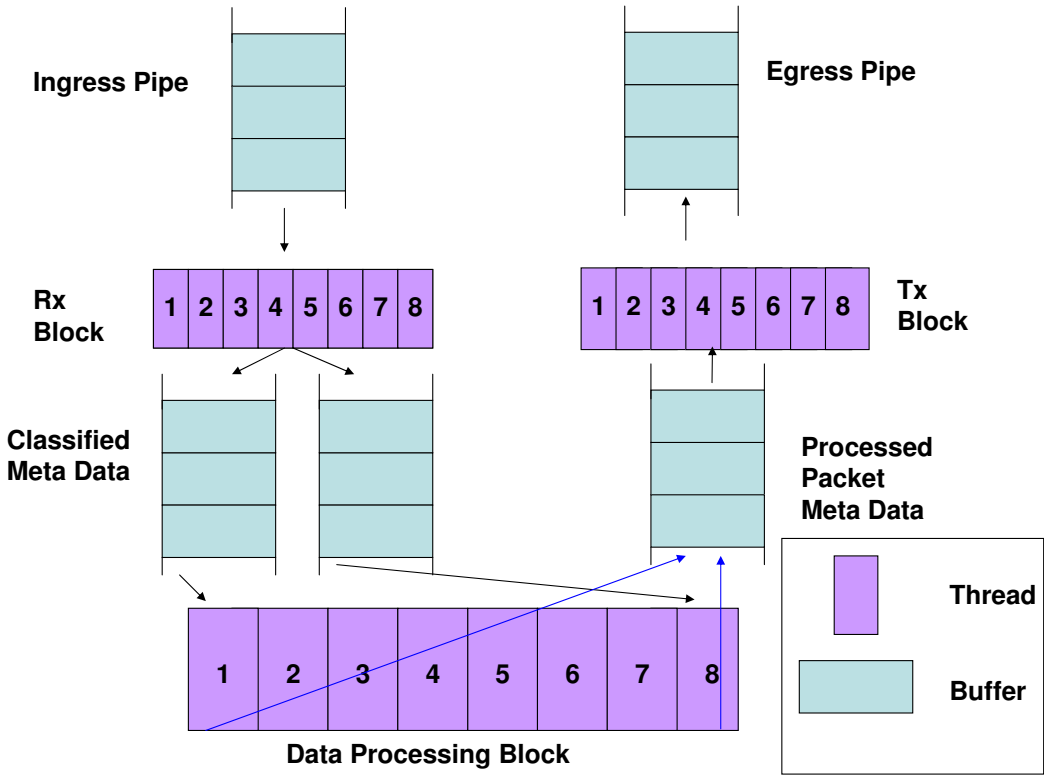

Figure 3: Logical Pipeline Structure

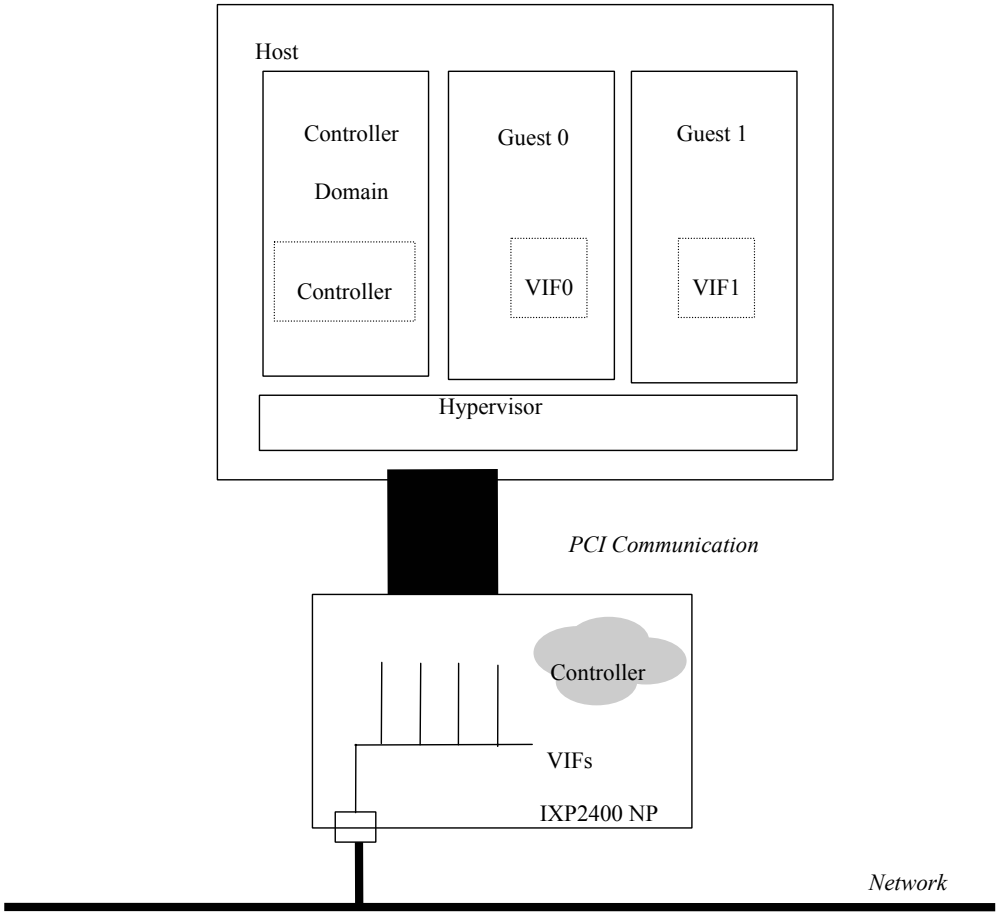

Figure 4: S-VNIC : Schematic Diagram [47]

we assume the existence of this support to next build in the QoS awareness.

The S-VNIC uses the information sent by guest VMs to compute scheduling policies and resource allocation requirements for all VNICs corresponding to all VMs, where resources include network processor resources, such as IXP microengine contexts, and memory resources available on the S-VNIC. This functionality is implemented in a similar fashion as described in Section 3.2 with small differences. Mainly, the meta data queues are software queues maintained in SDRAM of the IXP, which can be updated by the device driver and the appliance. Since the atomicity provided for enqueue and dequeue operations of hardware queues is no longer there, software

locking needs to be used. The update to the head and tail of the queues is done within critical sections. As usual there are separate Rx and Tx queues. The other difference is that while in Section 3.2 the prototype was tested with Rx receiving packets from the network, in this implementation, packets are generated by the host and are fed into the Rx queues. Quality awareness is implemented in the Host to network (IXP) communication and not in the reverse direction. The rest of the components described in Chapter 2 are implemented as described in Section 3.2. Part of this work is also envisioned and described separately in [46, 17]

3.4 Chapter Summary

In this chapter we described the IXP2400 on which we have implemented our prototype. We also described the realization of the implementation of the mechanisms described in Chapter 2. The other details of the implementation are closely associated with the results and hence will be introduced in the Chapter 4. Finally we describe the implementation of quality awareness in S-VNIC as an extension of the IQ-Appliance implementation. This helps in the realization of one use case for the developed functionality.

CHAPTER IV

EXPERIMENTAL RESULTS

In this chapter we discuss the results from the experimental analysis of the prototype IQ-appliance in order to support the claims and design made so far in the dissertation. We first start with the customary experimental setup. Our first set of results motivate the need and importance of runtime quality awareness. This helps us to choose the most suitable resource allocation scheme to continue with the rest of the experiments. We then evaluate the feasibility of providing for such a functionality in an information appliance. The next 2 sections discuss the capabilities introduced by such a functionality. We also analyse the affect of quality awareness on drop distribution in flows. Finally we compare a quality aware S-VNIC to a quality unaware S-VNIC to evaluate the utility of quality awareness in a virtualized infrastructure and also include results from testing the S-VNIC as an IQ-appliance.

4.1 Experimental Setup

The experiments described in this chapter are conducted on a cluster of 2850 Xeon nodes, each with an IXP2400 network processor card, interconnected via 1Gbps Ethernet. One of the IXP cards is used as the network appliance being evaluated, while other nodes are used as data sources and destinations. Since the hardware platform used in this prototype implementation is capable of handling much more than the 1Gbps data rates which we can deliver to it in our testbed, we artificially limit the capacities of the outgoing links to 500Mbps.

In the following experiments, 3 flows A, B, C suffice for demonstrating the notion of QoS 'built into' the appliance. The flows are assigned different amounts of runtime resources. For instance, flow A has a metadata buffer of 512 words which is twice the

size of B's and C's metadata queues. Each application flow also has some corresponding processing, which will be explained as and when necessary. The problem tackled below is to allocate computational resources to the different flows in the system, to ensure QoS guarantees at all points of time. This could also be viewed as a control system problem, where some feedback control is required to keep the system stable. In order to evaluate the QoS levels delivered to each flow, we vary the way in which the runtime allocates available resources, i.e., threads, to each flow, according to the following resource allocation schemes:

- Round Robin (**RR**) - Each thread polls the different flows in a round robin manner.
- Equal Static Allocation (**EAS**) - Threads are statically assigned to flows in a uniform manner.
- Static Allocation (**SA**) - Threads are statically allocated to flows according to priority levels.
- Priority Aware (**PA**) - Threads pull packets in the order of priority; lower priority flows and are serviced only when no higher priority packets exist.
- Priority Aware with Reservation (**PAR**) - A number of threads is preallocated to flows according to their priority (as in SA), while the remainder are shared in a priority-aware manner (as in PA).

In the experiments described in the remainder of this Chapter, flow B is always assigned the highest priority.

4.2 Importance of Runtime Quality-Awareness

4.2.1 The Case for Priority-Awareness

The first set of experiments evaluates the different policies for resource (i.e., threads) allocation to flows, where each flow consumes equal amounts of the ingress bandwidth

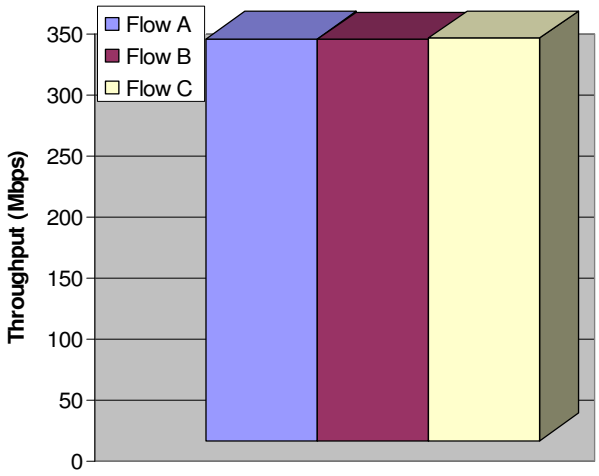

Figure 5: Equal Ingress Rates

(i.e., data items from each flow arrive at approx. 330Mbps as shown in Figure 5). The throughput measurements in Figure 6 show that while priority-sensitive schemes exhibit certain performance gains, the overall improvement in throughput levels is on the order of 10% for the high priority flow, and still almost 50% less than its incoming rate. Latency improvements, are however more pronounced, as can be seen in Figure 7 since data is dispatched more promptly form higher priority queues. These results demonstrate:

1. the importance of enabling some form of quality-awareness in the runtime; and
2. the need for greater coordination at multiple levels throughout the data path through the system.

4.2.2 The Need for Pre-Queuing IQ-Flow Handling

Next, we exploit the runtime ability to associate application specific behaviors (i.e., codes) with select changes in the runtime operating conditions. Such event triggers

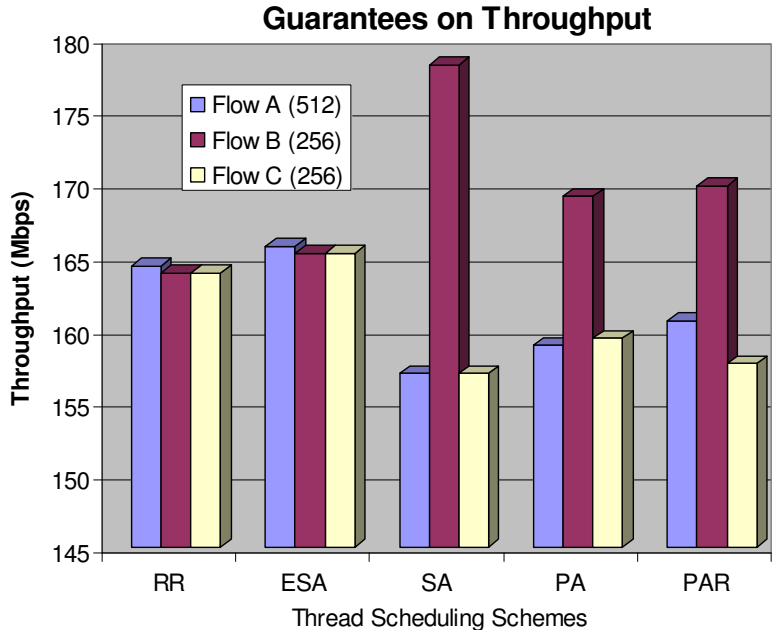

Figure 6: Throughput with Different Thread Allocation Policies

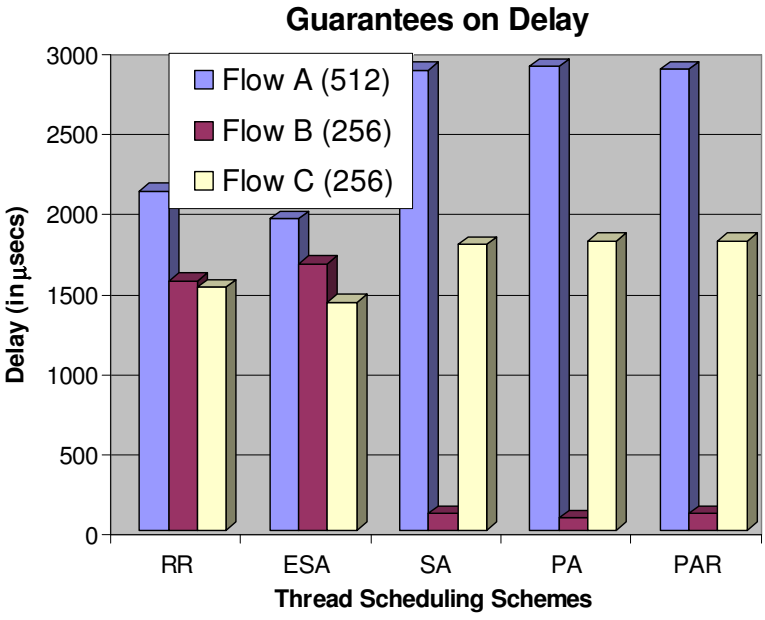

Figure 7: Delay with Different Thread Allocation Policies

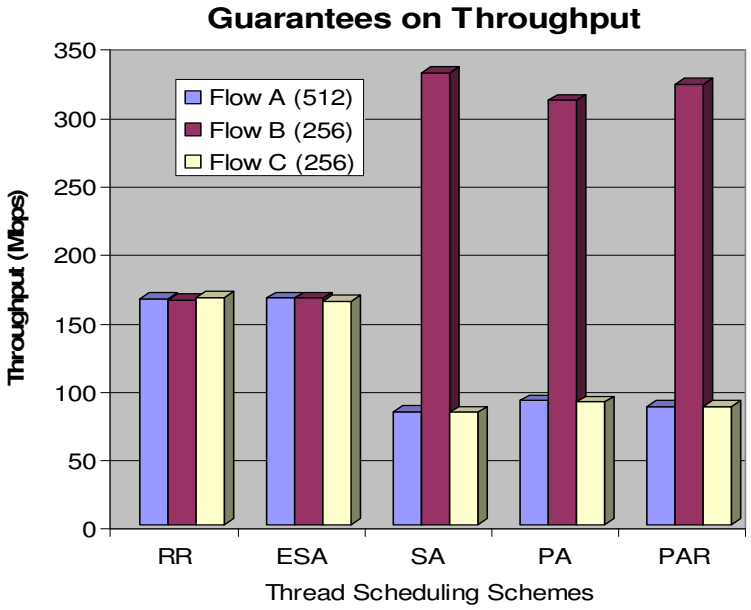

Figure 8: Throughput with Pre-Queuing/Ingress IQ-Flow Handling

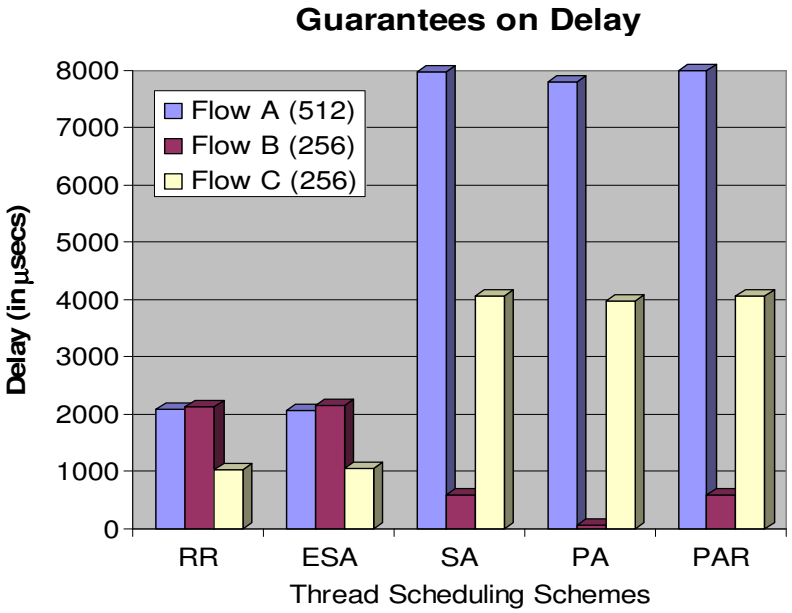

Figure 9: Delay with Pre-Queuing/Ingress IQ-Flow Handling

can be exploited to permit graceful degradation of the service level under overloaded conditions. For instance, we may want to apply application-specific filtering actions to the less important flows (e.g., drop data in A and C if the respective queues are full), in the event of a critical event occurrence in the high priority flow (i.e., B). Such filtering actions are most suitable for pre-queuing execution, and are associated with the Rx codes. The results in Figures 8 and 9 illustrate the throughput and latency levels observed under different platform configurations when such Rx-side execution of application-level handlers/codes is permitted. We refer to this as Priority-Aware Rx. We observe that throughput levels for the high priority queue finally reach the desired input levels. The graphs also demonstrate that providing platform events to which the application can respond in some custom manner, without having additional platform support to enforce different quality levels is insufficient (see bars for Priority Unaware RR or SA in both latency and bandwidth graphs.).

4.2.3 Importance of Resource Reservation

In order to guarantee certain minimal quality-levels, which are particularly important in virtualized environments, especially in the presence of dynamic variations in the workloads, it is necessary to rely on resource reservation. The results in Figures 11 and 12 are gathered for unequal input rates, with flow B's rate reaching 700Mbps, and flows A and C consuming the remaining 300Mbps on incoming bandwidth as shown in Figure 10. The results show that under spikes in the higher priority workload, a purely priority-driven runtime behavior may be optimal for the high priority flow/service, but it may result in unacceptably low service levels for the remaining flows.

We present the results only for Priority Aware(PA) and Priority Aware with Reservations (PAR) schemes. PA completely starves the lower priority flows in order to service the higher priority flows, and drastically increases the observed delays. Such

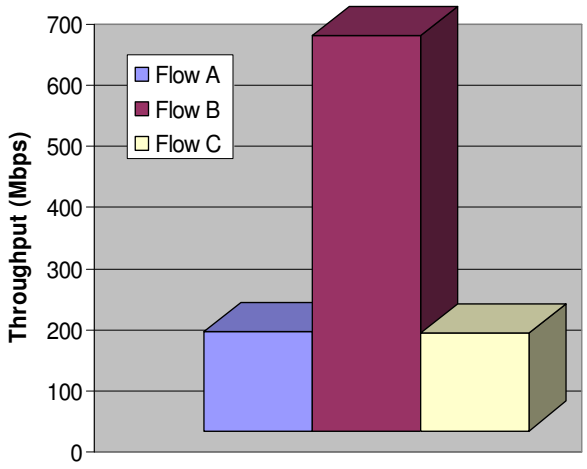

Figure 10: Aggressive Higher Priority Flow

purely priority based resource allocation policy is therefore unacceptable in the aforementioned environments.

Thus, we summarize the results as follows:

1. Priority-aware resource allocation schemes, such as SA, PA and PAR, are important for delivering differentiated service levels to incoming data flows.

2. Associating IQ-handlers with platform event triggers permits applications to dynamically install pre-queuing IQ-handlers, which further improve the platforms' ability to meet application-level quality requirements.

3. Coupling priority-aware resource allocations with mechanisms for per-flow minimal resource reservations is necessary to avoid unacceptable service degradation.

These observations cause us to focus the remaining experimentation on the SA and PAR schemes, as they are the only ones which provide for priority-based resource allocation.

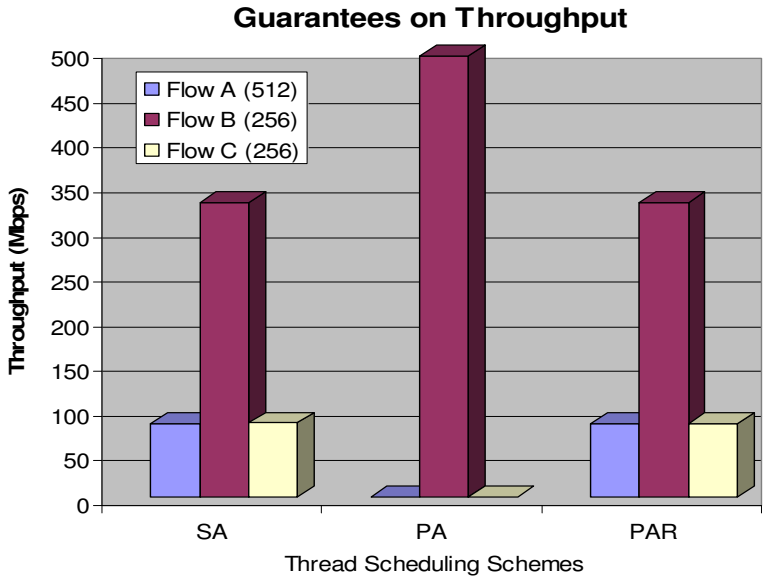

Figure 11: Importance of Minimal Resource Reservation : Throughput

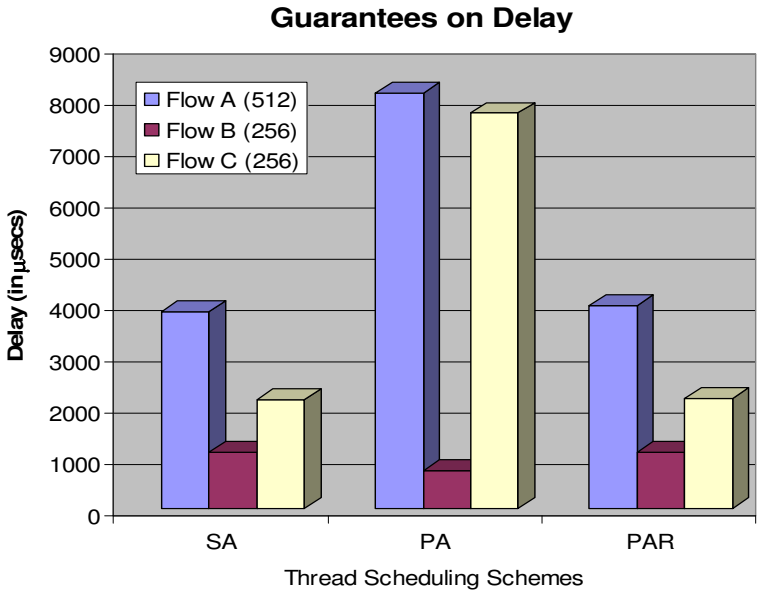

Figure 12: Importance of Minimal Resource Reservation : Delay

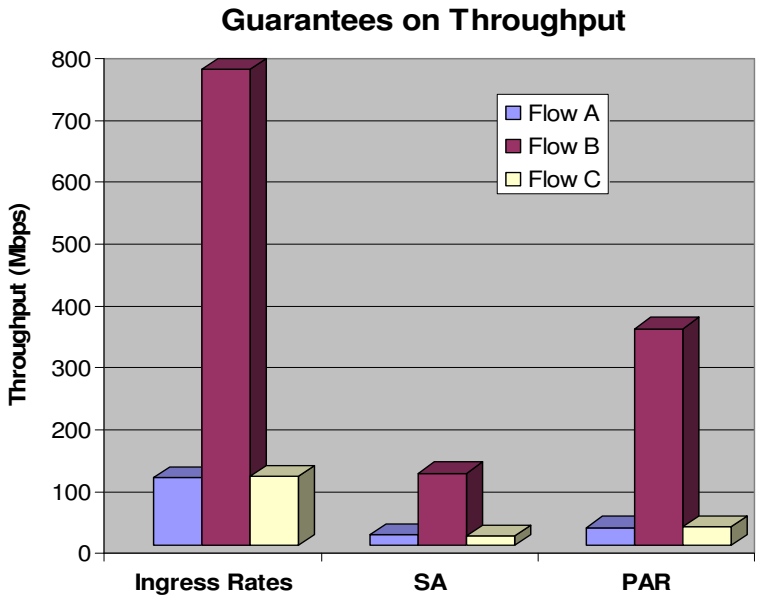

Figure 13: Importance of Dynamic Resource Allocation : Throughput

4.2.4 Improved Utilization of Platform Resources

Focusing on the trade-offs between dedicated resource usage and limits in the availability of shared resources, we see that the comparison of SA and PAR in Figure 13 indicates that with SA, the outgoing link reaches barely 40% utilization. In the second case, PAR ensures sharing of runtime resources, so that the outgoing link is fully utilized (500Mbps cumulative throughput). Interestingly, even the throughput levels for the lower priority flows experience a slight increase. Overall, with the PAR scheme, we attain more than a 300% improvement in throughput levels and more than a 50% latency reduction to the critical flow as can be seen in Figure 14.

4.3 Feasibility of Achieving Quality-Awareness

4.3.1 Monitoring Overheads

The next set of experiments aims to understand the overheads and impact of continuous platform monitoring. The resources being monitored are specified as part of the

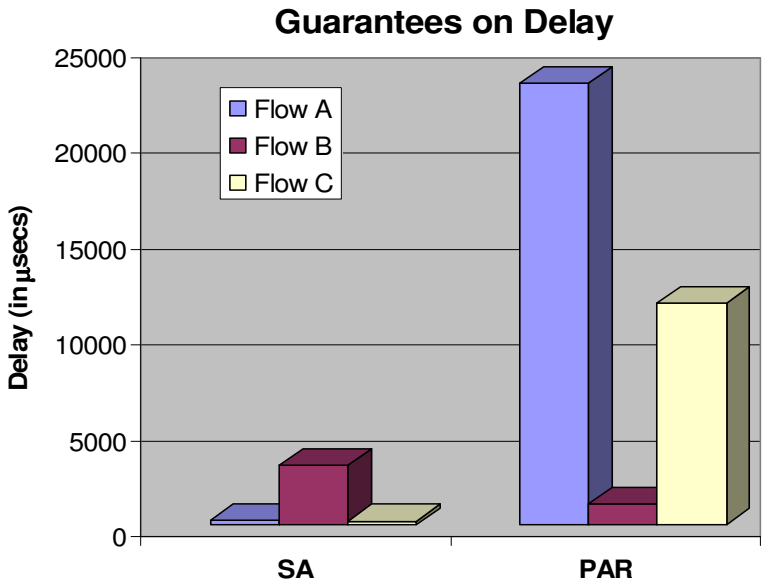

Figure 14: Importance of Dynamic Resource Allocation : Delay

runtime's configuration state, and as their size and type vary, so do the overheads. For simplicity, we distinguish between 'Simple' and 'Fancy' monitoring, which differ significantly in the number of runtime parameters being monitored. Simple monitoring tracks the aggregate resource utilization at a particular priority (i.e., QoS) level, whereas Fancy monitoring further differentiates the resources consumed by individual flows within a QoS level. In addition, we also support Queue Length Monitoring, which reports only the lengths of the different queues in the system. The results in Figure 15 indicate that even with Fancy Monitoring, when we monitor all runtime parameters, in the worst case, we observe a latency increase of at most 12.4% compared to the minimum packet delay for the highest priority flow. The overheads are significantly lower with respect to other flows and data sizes in the system.

Next, we evaluate the impact of varying monitoring frequency. For frequent monitoring, the priority specifications are checked at every context, i.e., distributed monitoring. For infrequent monitoring, a single control context is given the responsibility

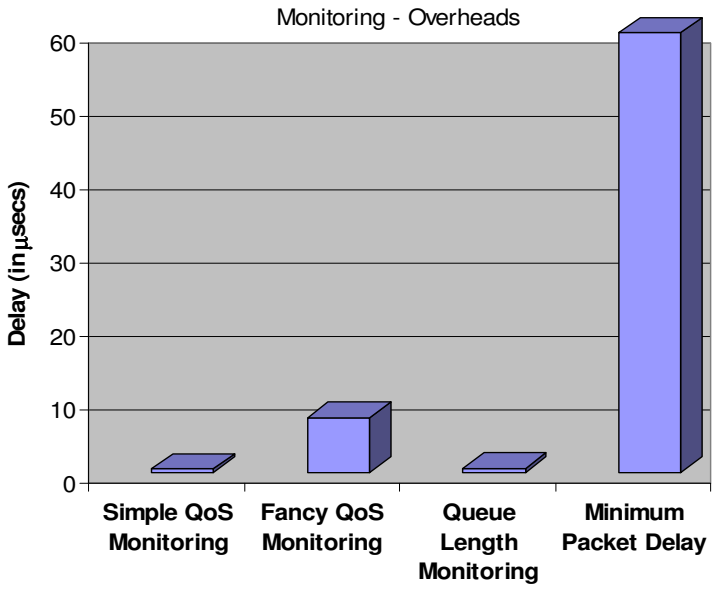

Figure 15: Monitoring Overheads

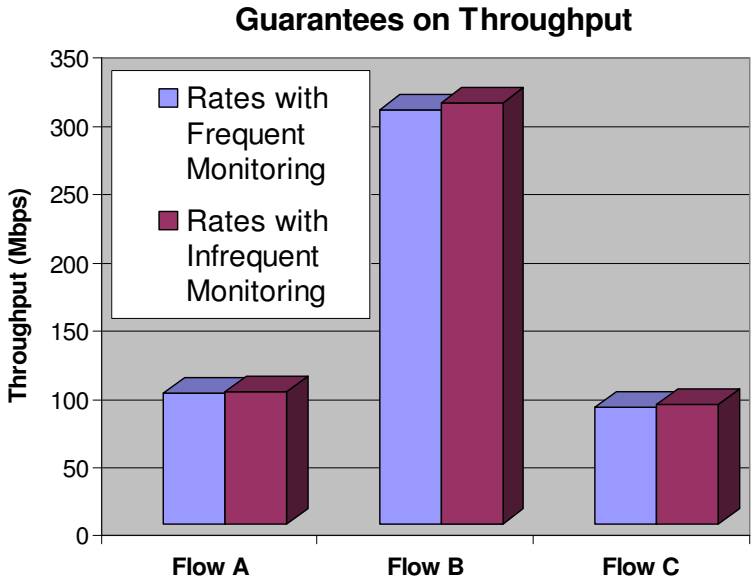

Figure 16: Monitoring Effects on Rates

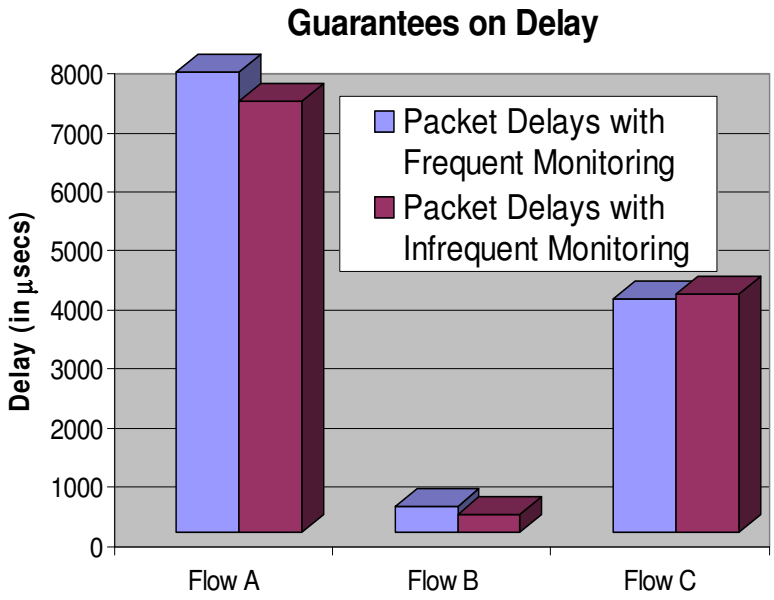

Figure 17: Monitoring Effects on Delay

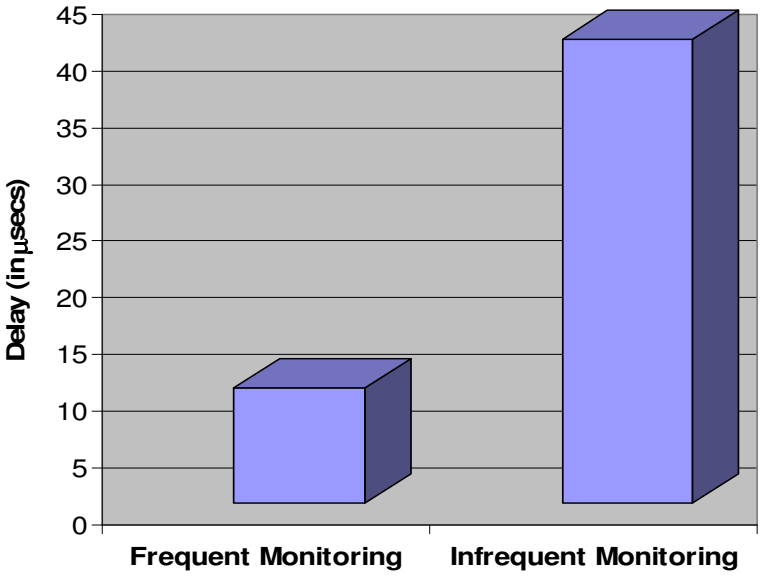

Figure 18: Monitoring Response Time

of reading the priority specification whenever it gets a chance to run. Figure 16 shows that the frequency impact on packet throughput is negligible and remains within few percent for different flow priorities. More interestingly, due to the parallel nature of the execution platform, and the hardware supported thread scheduling, which performs context switches only when threads perform I/O operations, increasing the monitoring frequency may even reduce message delays. The results in Figure 17 show negligible delay increases for one of the three flows (C) and more visible delay decreases for the other two flows. Finally, we also evaluate the runtime responsiveness to changes in some of the monitored parameters, as a function of the monitoring frequency. We observe that even with infrequent monitoring, we can detect changes in runtime parameters within $45usec$, which is less than 2/3 of the minimum packet delay. This is again due to the fact that the monitoring functionality is distributed across multiple hardware supported contexts in the parallel platform.

4.3.2 Classification Overheads

We also evaluate the overheads associated with other runtime mechanisms described in Section 2. The results in Table 1 show that accesses to configuration state distributed across DRAM, SRAM and Scratch can be executed with negligible overheads, of less than $0.3usec$ for state in DRAM, including the classification operation. Updates to individual state entries can be performed with maximum $1usec$ penalty (again, for DRAM-resident state), and the maximum observed delay to consistently propagate updates from the external configuration host to individual fast path processing thread is less than $100usec$.

4.4 Ability for Dynamic Adaptation

A set of experiments validates the platform's ability to dynamically adapt to changes in operating conditions. Figure 19 demonstrates the adaptability of the system. The bars marked Input track the ingress flow rates over time, and the interleaved bar sets

Table 1: Runtime Overheads

Operation	**Overhead**
Classify/DRAM	$0.256usec$
Classify/SRAM	$0.1usec$
Classify/Scratchpad	$0.095sec$
Update Single Entry	$1usec$
Maximum Consistency Delay for Updates from Single Host	$99.86usec$

marked with Dynamic Adaptation Scheme (DAS) report the observed egress rates. Initially, the resource allocation policy gives equal priority to all flows as long as it can keep up with their rates (as shown in the first pair of bars). As soon as the control core detects the ingress rates increases at time *t2*, it switches to a resource allocation policy of priority scheduling with reservations, so as to give maximum resources to the critical flow B, while still meeting the minimal QoS levels for the remaining flows (see the second pair of bars). As a result, flow B's data is processed at line rates, while the remaining flows receive minimal bandwidth levels. Furthermore, when B's ingress rate is reduced at *t3*, this policy results in appropriate adjustments so that the remainder of the available resources is allocated to flows A and C. Note that at all times, the maximum available outgoing bandwidth of 500Mbps is fully utilized. Similar behavior can be observed if multiple data streams are classified in flows.

4.5 Ability to Support Classes of Priority

The results in Figure 20 demonstrate that the resource scheduling policies implemented in our system can easily sustain multiple streams per flow class, and partition the available shared outgoing bandwidth appropriately across and within individual flows, i.e., classes.

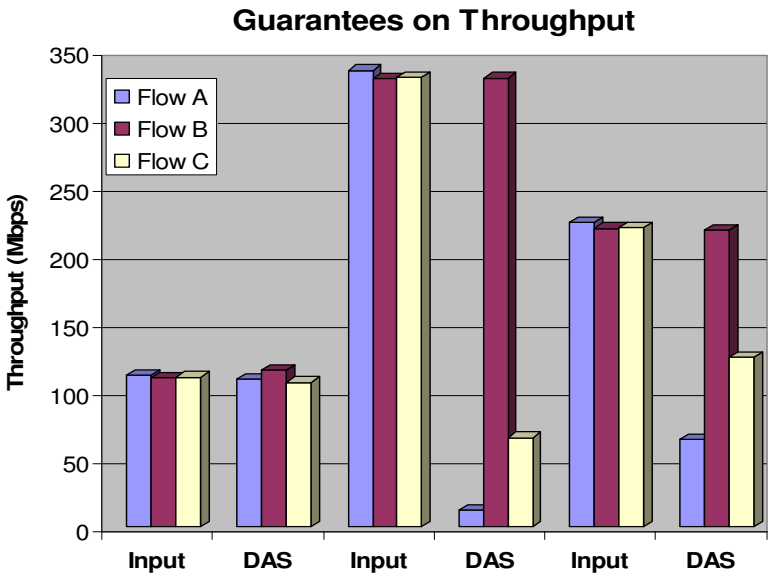

Figure 19: Adaptive Scheduling

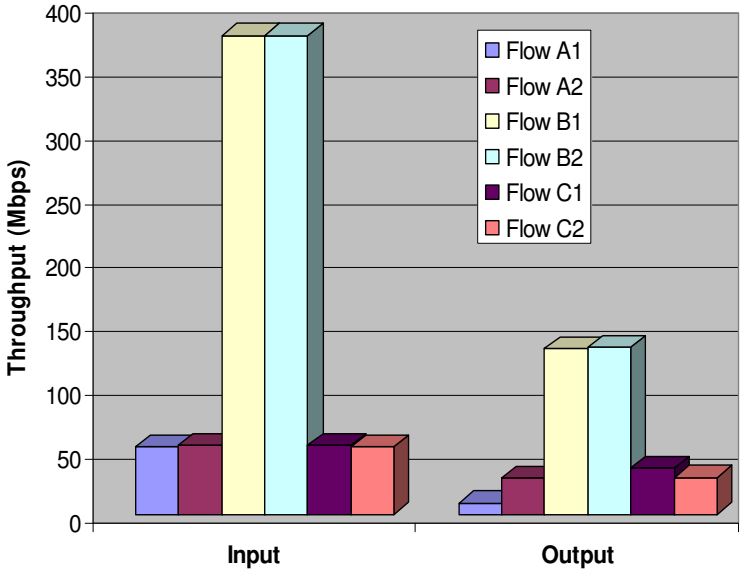

Figure 20: Classes of Priority

4.6 Multi-Instance Appliances

Our next set of results use replays of actual enterprise data, collected at one of our industry partners, Delta Air Lines. Here, three different processing actions, i.e., IQ-flow handlers, are applied to select data items, depending on their type. Consider data items from multiple sources at an airport that need to be exchanged with various system components. Flow A consists of updates to an external service, such as caterers. Delta data needs to be purged from sensitive information and its format standardized, resulting in 100B data item. Flow B consists of airport display updates. The processing and format translation actions for this data type are simpler, but of higher priority, and result in a 300B data item. Finally, flow C consists of state updates for fault-tolerance reasons, which result in a limited number of data accesses and minor size changes. The incoming data streams consist of a uniform distribution of all data types.

The results depicted in Figures 21 and 22 lead to two conclusions. First, they demonstrate the feasibility of supporting a mix of data handlers on a single 'multi-appliance' platform. Second, they illustrate that even under different processing requirements for individual flows, the proposed appliance design, where platform-level parameters are monitored and used to drive the selection of 'scheduling' actions, result in most desirable performance levels. We observe that the most important flow receives the highest percentage of available outgoing bandwidth, with lowest latencies. The remaining bandwidth is evenly distributed amongst updates from the other flows, with flow A experiencing larger per-data item delays due to the significantly greater complexity of the data handler itself. Additional experimentation (not reported here) with data streams consisting of larger data sizes, based on the Delta data and with similar handler behaviors, demonstrate that data sizes do not affect the platforms ability to continue to demonstrate the same desired behavior.

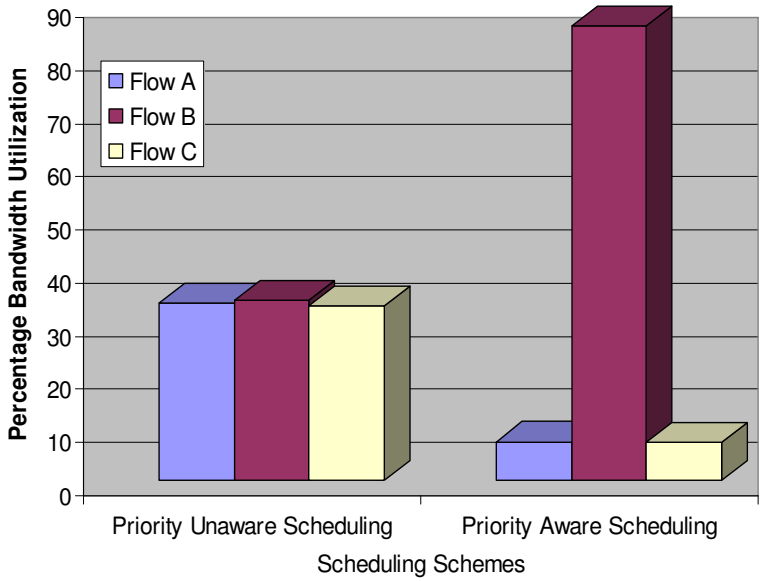

Figure 21: Bandwidth Utilization for Delta Airlines Data with Three Different IQ-Handlers

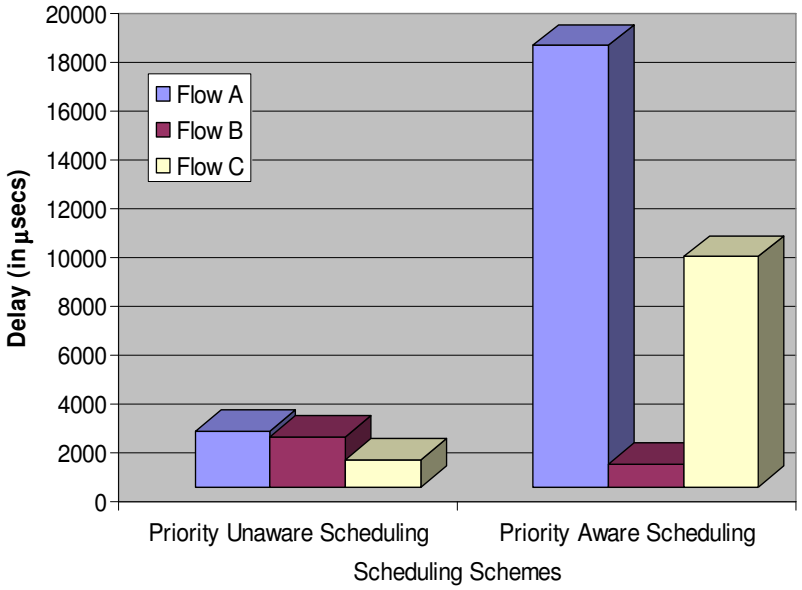

Figure 22: Delay for Delta Airlines Data with Three Different IQ-Handlers

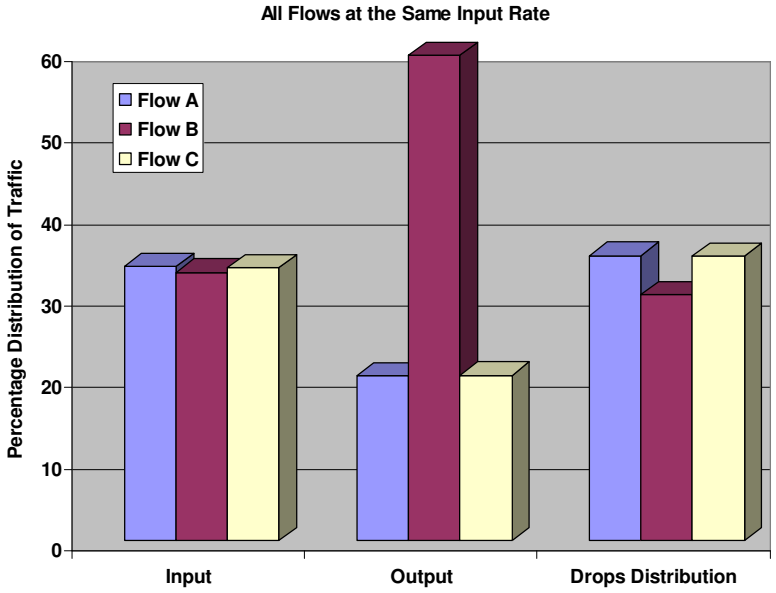

Figure 23: Drop Distribution for Equal Incoming rates

4.7 Analysis of Drop Distributions

So far we have compared throughput and delays in various scenarios. In this section we observe the result of prioritizing drops. Lower priority traffic is dropped early on in times of congestion. Figure 23 shows the effect of early on drops to make way for priority traffic. In case of an aggressive higher priority flow shown in Figure 24 the percentage of drops of packets from the higher priority flow increases due to filling up of the meta data and data buffers for the flow. We infer that these drops are involuntary drops due to limited resources since the higher priority flow itself if getting 90% of the egress throughput, and cannot go beyond this (since resources are reserved for lower priority flows to prevent starvation). Finally Figure 25 simulates the last scenario, where lower priority flows overwhelm the higher priority flow. Here the drops for the higher priority flow don't go beyond 12% while the lower priority traffic is dropped largely because of filled meta data buffers. However the drops never reaches a high of almost 30% as was reached in Figure 24. Such prioritization of drops

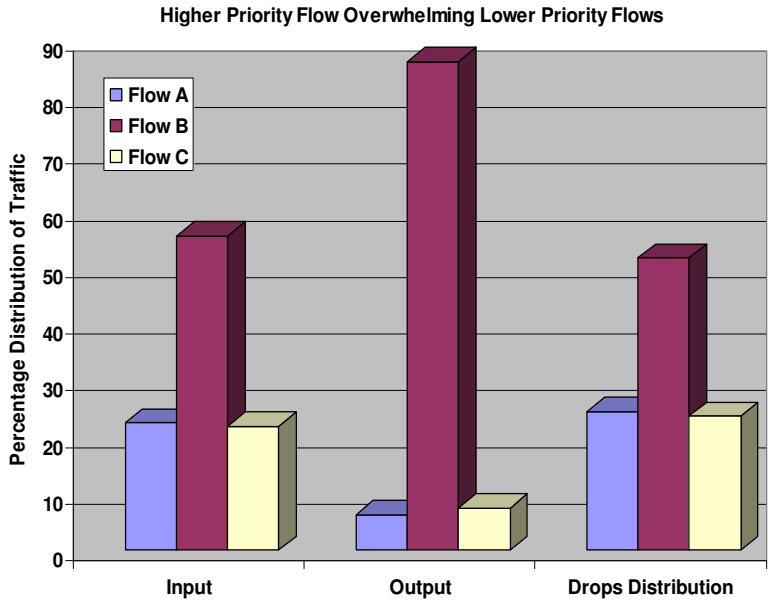

Figure 24: Drop Distribution for Aggressive Higher Priority Flow

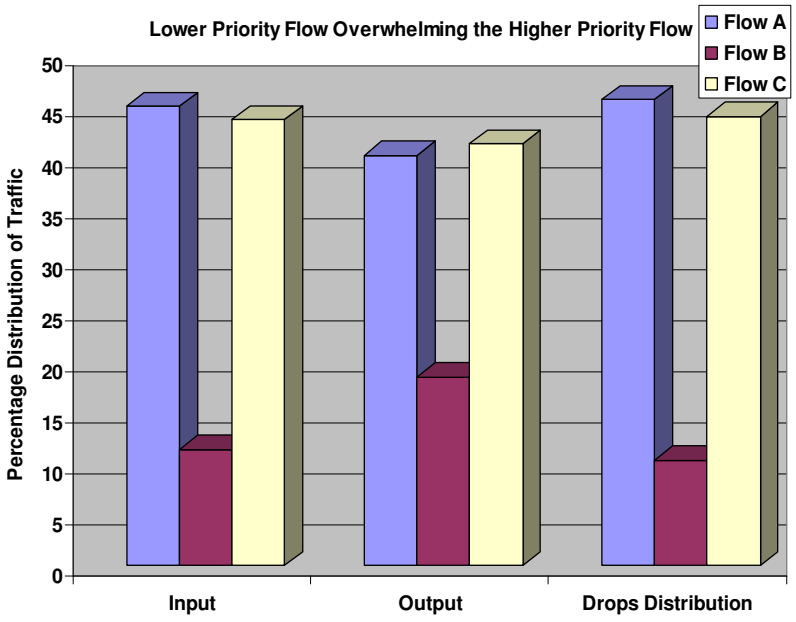

Figure 25: Drop Distribution for Aggressive Lower Priority Flows

may need to be built in for an application which can withstand drops but requires guaranteed throughput.

4.8 QoS Support for S-VNIC

Finally we evaluate the importance of QoS awareness in virtualized infrastructures. As seen in Figure 26 three VMs, with one VNIC each, are used to perform network I/O. One VM is set at high priority, and the other two are set at lower priority. The priority information is communicated to the S-VNIC, as described earlier in Section 3.2. The Quality-Aware (QA) S-VNIC correctly recognizes a situation of high input rate for all three flows and switches to providing guaranteed throughput to the indicated flow. In contrast, the Quality-Unaware (QU) S-VNIC lacks the functionality to discriminate across different QoS requirements, which results in all flows receiving a random percentage of the total egress throughput at any point in time.

The next Figure 27 shows the impact of introducing information processing to the S-VNIC. Each virtual flow is handled differently, with the highest priority flow having the most intensive information handler touching 100 bytes of application data. The lower priority flows have the S-VNIC inspecting 50 and 70 bytes of application data respectively. This result illustrates that the Quality Aware S-VNIC allocates as many resources as possible to the higher priority flow to keep up with the guarantees despite the demanding information handling function. As can be seen the total egress throughput has decreased from the previous graph due to the handlers. However the advantage of doing such information processing on the NIC as opposed to the host has been discussed in [32].

4.9 Chapter Summary

In this chapter, we developed an argument supporting the inclusion of quality awareness in appliances. For this we first presented a set of results which motivate the

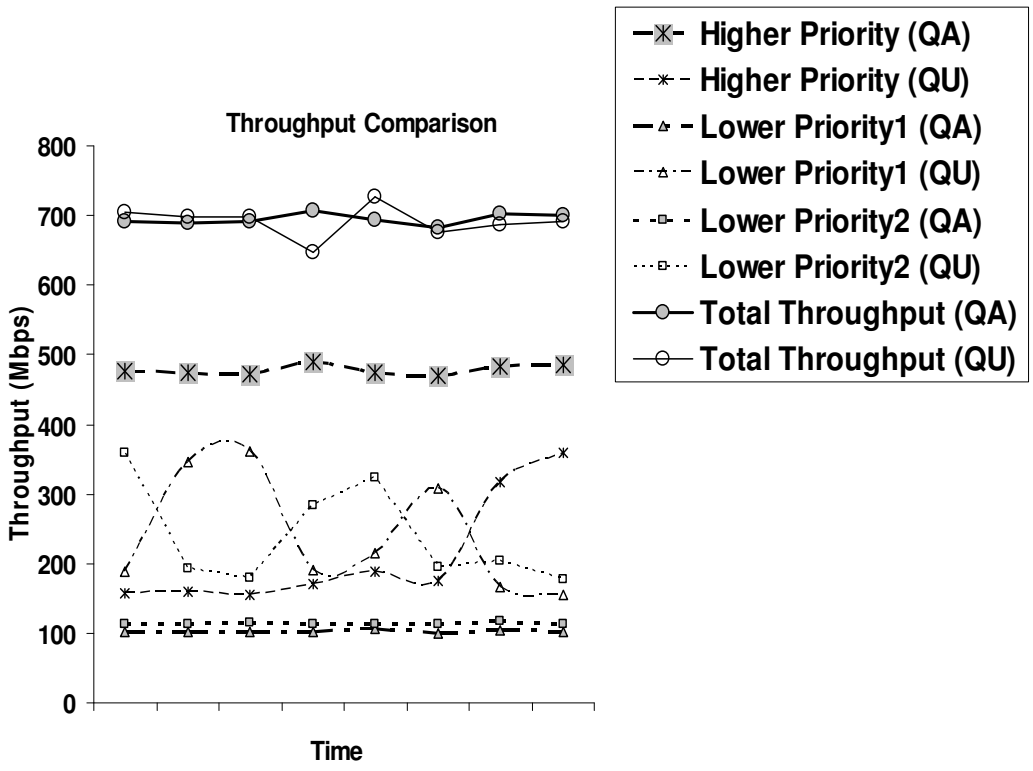

Figure 26: Comparison of Quality Aware (QA) S-VNIC with Quality Unaware (QU) S-VNIC

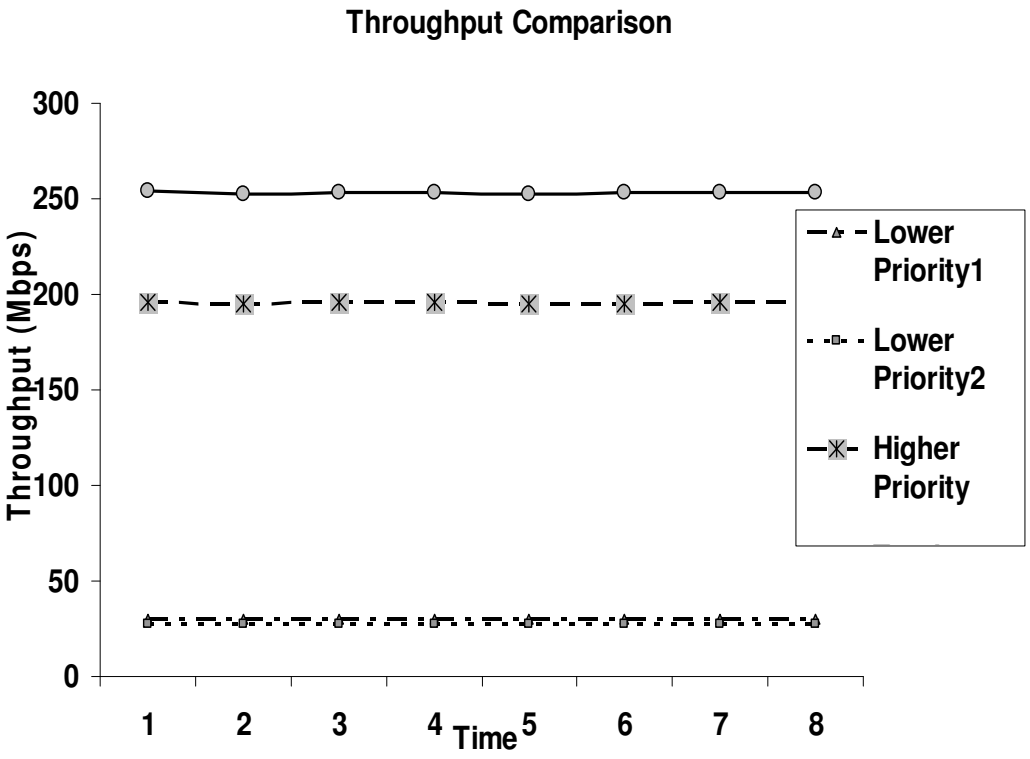

Figure 27: Semantically Enhanced Quality Aware S-VNIC

importance of runtime quality awareness. We found that not only does priority sensitive scheduling help provide guarantees but also improves the utilization of platform resources. Minimum resource reservation is found to be important to prevent starvation of lower priority flows. Next we evaluated the overheads due to monitoring and classification and found that the cost of these two operations is small compared to the system costs for processing the data flows. This makes us confident that quality awareness can be an important value addition to functionality of an appliance with low associated costs. Next we demonstrated the ability to have 'configurable scheduling' which responds to system input. The system design also supports different classes of service as in the classical QoS domain and multi-instance appliances. We concluded with the implementation of quality awareness in the S-VNIC to demonstrate a use case for the developed concepts. We established that a quality aware S-VNIC used as a logical device would provide more predictable and guaranteed performance than a quality unaware one. With this set of results we demonstrate the importance, feasibility and the utility of quality awareness in information appliances.

CHAPTER V

RELATED WORK

The objective of our work is to introduce quality awareness into information appliances to deal with dynamic application behaviors and the consequent changes in the resource needs of their data flows. For this we define an architecture for IQ-appliances and their runtimes which provides online monitoring, resource allocation, and adaptation needed by higher level quality management methods. The objective and the approach we use is influenced by several research directions in the past. It also relates to several past and ongoing research efforts by context, level at which solutions are developed, target platforms or application domain addressed. This chapter surveys the related work, by classifying it into several groups based on specific similarities with the work presented in this thesis.

In the rest of the chapter we categorize the related work as follows. We first provide research related to Extensible Network Infrastructures - viz. Active Networks, Device Level Research and Use of Network Processors for Application Specific Services. Information Virtualization is motivated for similar reasons as this research. We then identify research related to Information Virtualization, Virtualization, Multicore Architecture and Monitoring and then move on to Chapter Summary.

5.1 Extensible Network Infrastructures

5.1.1 Active Networks

Our work addresses appliances which transform / inspect data flows on the fly. Such in-transit data manipulation services have been addressed by a large body or work, from active networking research [58]. Research with active networks has explored the possibilities of extending and customizing the behaviour of the network infrastructure

to meet application needs [4, 6, 52]. Furthermore, The Active System Area Networks (ASAN) [24, 38] focuses on the use of NPs close to or even within the leaf nodes of a network. Information appliances are similar to active network infrastructure in the sense that they aim to exploit the programmability in the networking infrastructure and to dynamically associate application specific codes with select data flows. However the difference is that these appliances are employed at network edges.

5.1.2 Device Level Research

The work presented in this dissertation builds on our previous research to create integrated platforms of hosts and attached network processors (NPs), so as to enable the execution of application-specific services onto the programmable NP and closer to the network [16]. [32] builds in database operations into the NP to simulate a heterogeneous multicore infrastructure. The operations carried defined in [32] are also used in this work to prototype an information appliance. [48] describes the implementation of a self-virtualized NIC which provides a virtual interface to the VM above. Again operations described in [32] integrated with [48] provided us with a logical device. This work differs from the above mentioned work in that it provides QoS aware resource allocation / scheduling within the appliance so that it can provide guarantees to applications.

Our work at the appliance level can also be related to efforts on augmenting the functionality of communications devices for select application tasks [50, 36]. There are numerous interface designs for closely coupled network devices such as programmable network processors or line cards to host nodes using OS controlled mappings [52, 53, 8]. The motivation is to take advantage of the network near nature of these devices vs. hosts and implement transaction services, synchronization functions and service or system specific optimization of protocol stacks and of the data movement and buffering associated with message communication. Similar argument

is made for use of specialized devices for attaining intelligent disks that can execute application specific I/O functionality with much improved performance levels over standard hosts [30, 1, 31]. IQ-appliances intend to provide performance benefits over that of information appliances by making them priority sensitive.

In other work from our group [57] describes an algorithm which can be used for delay /deadline sensitive traffic through a NP. Our work differs from this work in that scheduling policies are included as part of runtime components of the architecture proposed in our work to achieve QoS awareness. The scheduling itself is reconfigurable depending upon system parameters at any point of time particularly in keeping with application needs.

5.1.3 Use of Network Processors for Application Specific Services

The utility of executing compositions of various protocol- vs. application-level actions in different processing context is already widely acknowledged. Examples include splitting the TCP/IP stack across general purpose processors and dedicated network devices, such as network processors, FPGA-based line cards, or dedicated processors in SMP systems [9, 49, 5], or splitting the application stack as with content-based load balancing for an HTTP server [64]. The programmability of network processors has been widely exploited in both industry and academia, for delivering more flexible network and application-level services [23, 62]. Integrated host-NP platforms could exemplify future heterogeneous, multi-core systems. Similarly, in modern interconnection technologies, the network interfaces represent separate processing contexts with capabilities for protocol off-load, direct data placement, and OS-bypass [63, 39]. IQ-appliance goes a step further as to not only offload functionality from the data intensive application but also couple knowledge of priorities with the General Purpose Processor, and achieve intended performance levels for important data flows. Finally, IXP based improvements for wide area applications are attained by enabling packet

header based customization of an incoming data stream, thereby offering services such as software routing network monitoring etc. The DiffServe IXP server allows applications to specify the service classes differentiated by the server in an application specific manner [35]. This approach relies solely on network level information embedded in packet headers to perform the specific functionality. In our case also, the packet headers which have different IP addresses in case of implementation on S-VNIC or other identifiers in case of th IQ-Appliance are used to classify the flows before scheduling them based on priority. The difference here is that the appliance is an information appliance nd we use monitoring of system resource to adapt to variations in load and build a closed feedback loop.

5.2 Information Virtualization

The ability to modify the original data stream, and personalize it, or according to a set of criteria, customize it for its final destination, has been previously termed data or information virtualization [56]. This work uses this term to denote application-specific data handling. As mentioned before in-transit transformation on data flows is also addressed in [58].

5.3 Virtualization

One specific setting in which honoring individual QoS requirements is particularly important is in virtualized environments. Virtualization techniques have received recently much attention at the network [44, 21], CPU [7, 15], or device [48, 37] level, mainly as a technique to consolidate different workloads, and permit their efficient sharing of a single infrastructure. While the primary virtualization objective in these efforts is to provide isolation, our focus is on providing runtime support for efficient sharing of the platform resources, while still meeting per-flow QoS guarantees. Similar approaches have been used in other domains, such as techniques for reservation and sharing of cluster server or network resources [13] our focus in on the resources of an

individual many-core platform.

5.4 Multicore Architecture

The need for QoS for applications multiplexed over several cores, their corresponding cache, memory hierarchy and attached I/O devices are being addressed current research trends. [61] discusses need for QoS guarantees for data streams while [28, 29] discuss the need for the QoS guarantees with regard to caching. While this trend is relatively new, it is similar to QoS guarantees for appliances, and it might be possible to adopt the mechanisms suggested in this work to address such issues.

5.5 Monitoring and Adaptation

Finally, the runtime capabilities advocated in this work rely on continuous platform monitoring and adaptation. Similar techniques have been widely used for tuning and monitoring distributed enterprise systems [2, 3], to deploying codes in wide area infrastructures [33], to adaptive QoS for delivering scalable media to end systems [11], etc. Our work differs from this work in that monitoring is a part of suggested runtime components of the architecture. We mainly focus on the components of the framework, and intend that the monitoring is made reconfigurable.

5.6 Chapter Summary

We are not aware of other research on QoS Awareness in Information Appliances. However there has been a lot of work leading to information appliances, QoS aware monitoring, end-to-end virtualized systems and extensible network infrastructures. In this chapter we summarized the different research directions which are in one way or the other related to our own research. We next move on to concluding remarks.

CHAPTER VI

CONCLUDING REMARKS

In this chapter we summarize the main contributions of this work and then elaborate on future directions which could be explored.

6.1 Contributions

This thesis describes a runtime environment for the important class of 'information appliances' typically found in modern distributed infrastructures. The goal is to provide QoS guarantees to the data flows in appliances used by data-intensive applications and in virtualized environments. Additionally, dynamic adaptation and platform utilization are also addressed. The approach taken:

1. enables the creation of extensible appliances that can be customized to execute diverse application-specific processing actions;

2. enriches the runtime with QoS-aware functionality, IQ-appliances, so that the specific quality requirements of individual data flows can be met;

3. includes continuous monitoring and adaptation capabilities so as to optimize platform utilization and attain individual service levels, while still honoring flows' minimum quality guarantees; and

4. provides an example of such a QoS aware appliance by building it into self-virtualized devices.

We argue that the use of such appliances is particularly important for enterprise systems and applications, where increasing levels of system virtualization are creating an ever-increasing number of information flows with different needs and of different

types. Additionally we claim that these mechanisms can be applied in a broader context, such as in providing QoS guarantees in runtimes for multicore environments and network processors. We demonstrate that it is feasible to provide runtime support for QoS in appliances with relatively low overheads for marked gains in the way flows are isolated and serviced depending on application requirements. We further justify this claim by showing results from the implementation of these mechanisms in the S-VNIC and the benefit to applications running in the host above. Above all, we show that scheduling policies can be dynamically changed in order to suit the input rates and applications dynamically at any point of time. This is possible due to the monitoring component, which has low overhead. If configured properly, this would move us a step closer to autonomic systems.

6.2 Future Directions

This thesis opens several opportunities for future research directions. Some are directly connected to the present status of IQ-Appliances while others are related to the investigation of issues raised by IQ-Appliances in other context. Mainly the immediate next steps in research surrounding IQ-Appliances could be to understand what is the capability of a system which employs IQ-Appliances along with an OS-Bypass. Different scheduling policies on the host and corresponding policies on the appliance will provide insights into the coupling required. Implementing different appliances and understanding the best way to schedule different data transformations can also provide insights into scheduling policies suitable for IQ-Appliances.

Some of the more open problems related to this work are to understand what are the parallel requirements for QoS differentiation and isolation in other environments, for example multicore environments and mapping the suggested mechanisms in these domains. Furthermore, architectural enhancements to improve support for reconfigurable monitoring, resource allocation and atomic locking in appliances would help

improve the proposed runtimes. Finally compiler support to help the application to appliance coupling would help to improve the ease of integrating IQ-Appliances into real systems.

REFERENCES

[1] Acharya, A., Uysal, M., and Saltz, J., "Active disks: programming model, algorithms and evaluation," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, (San Jose, CA), 1998.

[2] Agarawalla, S., Eisenhauer, G., and Schwan, K., "Lightweight Morphing Support for Evolving Middleware Data Exchanges in Distributed Applications," in *Proc. of International Conference on Distributed Computing Systems*, 2005.

[3] Agarwala, S., Chen, Y., Milojicic, D., and Schwan, K., "QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems," in *The 3rd IEEE International Conference on Autonomic Computing*, 2006.

[4] Agere Systems, *The Case for a Classification Language. White Paper*, 2003.

[5] Albrecht, C., Foag, J., Koch, R., and Maehle, E., "DynaCORE-A Dynamically Reconfigurable Coprocessor Architecture for Network Processors," in *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2006.

[6] Alexander, D. S., Menage, P. B., Keromytis, A. D., Arbaugh, W. A., Anagnostakis, K. G., and Smith, J. M., "The Price of Safety in An Active Network," *Journal of Communications and Networks (JCN), special issue on programmable switches and routers*, vol. 3, pp. 4–18, Mar. 2001.

[7] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the Art of Virtualization," in *Proc. of 18th Symposium of Operating Systems Principles (SOSP-18)*, (Bolton Landing, NY), 2003.

[8] Bhattacharjee, B., Ammar, M., Zegura, E., Shah, V., and Fei, Z., "Application-Layer Anycasting," in *Proceedings of INFOCOM'97*, (Kobe, Japan), 1997.

[9] Braun, F., Lockwood, J., and Waldvogel, M., "Protocol wrappers for layered network packet processing in reconfigurable networks," *IEEE Micro*, vol. 22, pp. 66–74, Jan./Feb. 2002.

[10] Bustamante, F., Eisenhauer, G., Schwan, K., and Widener, P., "Efficient Wire Formats for High Performance Computing," in *Proc. of Supercomputing 2000*, (Dallas, TX), Nov. 2000.

[11] Campbell, A. and Coulson, G., "QoS Adaptive Transports: Delivering Scalable Media to the Desk Top," in *IEEE Network Magazine*, 1997.

[12] "Cavium networks." www.cavium.com, Jan. 2007.

[13] Chase, J., Grit, L., Irwin, D., Moore, J., and Sprenkle, S., "Dynamic Virtual Clusters in a Grid Site Manager," in *Twelfth International Symposium on High Performance Distributed Computing*, 2003.

[14] "Dark data center design." www.processor.com/editorial/article.asp?article=articles/P2835/31p35/31p35.asp, April. 2007.

[15] "The vmware esx server." www.vmware.com/products/esx/, Jan. 2007.

[16] Gavrilovska, A., *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors*. PhD thesis, Georgia Institute of Technology, 2004.

[17] Gavrilovska, A., Kumar, S., Raj, H., Schwan, K., Nathuji, R., Gupta, V., Niranjan, R., Randive, A., and Saraiya, P., "High-Performance Hypervisor Architectures: Virtualization in HPC Systems," in *Workshop on High Performance Virtualization (HPCVirt) in conjunction with EuroSys 2007*, Mar. 2007.

[18] Gavrilovska, A., Kumar, S., Sundaragopalan, S., and Schwan, K., "Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications," in *15th Int'l Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'05)*, (Skamania, WA), 2005.

[19] Gavrilovska, A., Mackenzie, K., Schwan, K., and McDonald, A., "Stream Handlers: Application-specific Message Services on Attached Network Processors," in *Proc. of Hot Interconnects 10*, (Stanford, CA), Aug. 2002.

[20] Gavrilovska, A., Schwan, K., and Oleson, V., "Practical Approach for Zero Downtime in an Operational Information System," in *Proc. of 22nd International Conference on Distributed Computing Systems*, (Vienna, Austria), July 2002.

[21] "Global environment for network innovations." www.geni.net, Jan. 2007.

[22] Gill, C. D., Kuhns, F., Levine, D., Schmidt, D. C., Doerr, B. S., , and Schantz, R. E., "Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems," in *Proceedings of the 1st International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, (Phoenix, Arizona), Nov. 1999.

[23] Guo, J., Yao, J., and Bhuyan, L., "An Efficient Packet Scheduling Algorithm in Network Processors," in *IEEE Infocom*, 2005.

[24] Hughes, E., MacKenzie, K., Schimmel, D., Schwan, K., and Yalamanchili, S., "Active system area networks for data intensive applications," in *Department of Energy PI Meeting*, Oct. 1999.

[25] Intel Corporation, *Intel Network Processor Family.* http://developer.intel.com/-design/network/producs/npfamily/, Jan. 2006.

[26] Intel Corporation, *Intel IXP2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications. White Paper*, 2003.

[27] Intel Corporation, *Intel IXP2400 Network Processor - Hardware Reference Manual*, Oct. 2004.

[28] Iyer, R., "Cqos: A framework for enabling qos in shared caches of cmp platforms," in *International Conference on Supercomputing*, June 2004.

[29] Jin, L., Lee, H., and Cho, S., "A flexible data to l2 cache mapping approach for future multicore processors," in *Memory Systems Performance and Correctness*, Oct. 2006.

[30] Keeton, K., Patterson, D. A., and Hellerstein, J. M., "A case for intelligent disks (idisks)," *SIGMOD Record*, vol. 27, Sept. 1998.

[31] Krishnamurthy, R., Schwan, K., and Rosu, M., "A Network Co-Processor-Based Approach to Scalable Media Streaming in Servers," in *Proc. of International Conference on Parallel Processing (ICPP)*, Aug. 2000.

[32] Kumar, S., Gavrilovska, A., Schwan, K., and Sundaragopalan, S., "C-Core: Using Communication Cores for High Performance Network Services," in *Proc. of 4th Int'l Conf. on Network Computing and Applications (IEEE NCA05)*, (Cambridge, MA), 2005.

[33] Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K., "Resource-Aware Distributed Stream Management using Dynamic Overlays," in *Proc. of 25th IEEE International Conference on Distributed Computing Systems (ICDCS-2005)*, (Columbus, OH), 2005.

[34] Kumar, V., Mansour, M., Martin, B., Mehalingham, J., and Schwan, K., "Policies for Enterprise Information Systems: Two Case Studies," in *10th IFIP/IEEE Symposium on Integrated Management*, (Munich, Germany), May 2007.

[35] Lin, Y.-D., Lin, Y.-N., Yang, S.-C., and Lin, Y.-S., "DiffServ over Network Processors: Implementation and Evaluation," in *Proc. of Hot Interconnects 10*, (Stanford, CA), Aug. 2002.

[36] Liu, J., Wu, J., Kini, S. P., Wyckoff, P., and Panda, D. K., "High Performance RDMA-Based MPI Implementation over InfiniBand," in *Int'l Conference on Supercomputing (ICS '03)*, 2003.

[37] Liu, J., Abali, B., Huang, W., and D.K.Panda, "Virtualizing InfiniBand in Xen," in *Xen Summit*, 2006.

[38] Mackenzie, K., Shi, W., McDonald, A., and Ganev, I., "An Intel IXP1200-based Network Interface," in *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*, (Anaheim, CA), Feb. 2003.

[39] Mehra, P., "Apsara: The Quest for the Perfect Server for Network Computing Applications," in *Proc. of Network Computing and Applications (NCA), keynote address*, (Cambridge, MA), 2003.

[40] "Network content flow analysis and virtual machine technologies solutions from netronome." www.netronome.com, Jan. 2007.

[41] Niranjan, R., Gavrilovska, A., and Schwan, K., "Towards IQ-Appliances: Quality-awareness in Information Virtualization," Tech. Rep. GIT-CERCS-07-06, Georgia Institute of Technology, 2007.

[42] Niranjan, R., Gavrilovska, A., Schwan, K., and Tembey, P., "Towards IQ-Appliances: Quality-awareness in Information Virtualization," in *The 6th IEEE International Symposium on Network Computing and Applications*, July 2007.

[43] Oleson, V., Schwan, K., Eisenhauer, G., Plale, B., Pu, C., and Amin, D., "Operational Information Systems - An Example from the Airline Industry," in *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[44] Peterson, L., Shenker, S., and Turner, J., "Overcoming the Internet Impasse through Virtualization," in *Proc. of HotNets-III*, (Cambridge, MA), Nov. 2004.

[45] RadiSys Corporation, *Radisys ENP-2611 Data Sheet*. http://www.radisys.com/-files/ENP-2611_07-1236-02_0803.pdf, Jan. 2007.

[46] Raj, H., Kumar, S., Niranjan, R., Gavrilovska, A., and Schwan, K., "iConnect: High Performance Semantic Communications for Virtual Machines," in *15th Annual IEEE Symposium on High-Performance Interconnects (Hot Interconnects)*, 2007. Submitted.

[47] Raj, H. and Schwan, K., "Implementing a Scalable Self-Virtualizing Network Interface on an Embedded Multicore Platform," in *Workshop on Interaction between Operating System and Computer Architecture*, 2005.

[48] Raj, H., Schwan, K., Agarwala, S., and Xenidis, J., "Scalable I/O Virtualization via Self-Virtualizing Devices," Tech. Rep. GIT-CERCS-06-02, Georgia Institute of Technology, 2006.

[49] Regnier, G., Minturn, D., McAlpine, G., Saletore, V., and Foong, A., "ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine," in *Proc. of Symposium of Hot Interconnects*, (Stanford, CA), 2003.

[50] Rosu, M.-C., Schwan, K., and Fujimoto, R., "Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture," *Cluster Computing, Special Issue on High Performance Distributed Computing*, May 1998.

[51] Royal, P., Halpin, M., Gavrilovska, A., and Schwan, K., "Utilizing Network Processors in Distributed Enterprise Environments," in *5th Int'l Conf. on Network Computing and Applications*, (Cambridge, MA), 2006.

[52] Taylor, D. E., Lockwood, J. W., Sproull, T. S., Turner, J. S., and Parlour, D. B., "Scalable IP Lookup for Programmable Routers," in *Proc. of IEEE Infocom 2002*, (New York, NY), June 2002.

[53] Taylor, D. E., Turner, J. S., and Lockwood, J. W., "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers," in *Proc. of 4th IEEE Conference on Open Architectures and Network Programming (OPENARCH'01)*, (Anchorage, AK), Apr. 2001.

[54] Tibco Software Inc., *Tibco ActiveEnterprise: XML Tools*. http://www.tibco.com/solutions/products/extensibility/, Jan. 2007.

[55] "Server virtualization and virtualization infrastructure management solutions from virtualiron." www.virtualiron.com, Jan. 2007.

[56] Weng, L., Agrawal, G., Catalyurek, U., Kurc, T., Narayanan, S., and Saltz, J., "An Approach for Automatic Data Virtualization," in *HPDC*, 2004.

[57] West, R. and Schwan, K., "Dynamic Window-Constrained Scheduling for Multimedia Applications," in *Proc. of 6th International Conference on Multimedia Computing and Systems (ICMCS'99)*, (Florence, Italy), June 1999.

[58] Wetherall, D. J., "Active Network Vision and Reality: Lessons from a Capsule-based System," in *Proc. of he 17th ACM Symposium on Operating System Principles (SOSP'99)*, (Kiawah Island, SC), Dec. 1999.

[59] Wiseman, Y., Schwan, K., and Widener, P., " Efficient End-to-End Data Exchange Using Configurable Compression," in *Proc. 24th International Conference on Distributed Computing Systems*, (Tokyo, Japan), 2004.

[60] Wolf, M., Abbasi, H., Collins, B., Spain, D., and Schwan, K., "Service Augmentation for High End Interactive Data Services," in *Proceedings of Cluster*, 2005.

[61] Yeh, T. Y. and Reinman, G., "Fast and fair: Data-stream quality of service," in *Conference on Compilers, Architectures and Aynthesis for Embedded Systems*, Sept. 2005.

[62] Yocum, K. and Chase, J., "Payload Caching: High-Speed Data Forwarding for Network Intermediaries," in *Proc. of USENIX Technical Conference (USENIX'01)*, (Boston, Massachusetts), June 2001.

[63] Zhang, X., Bhuyan, L. N., and chun Feng, W., "Anatomy of UDP and M-VIA for Cluster Communications," *Journal on Parallel and Distributed Computing, Special Issue on Cluster and Grid Computing*, 2005.

[64] Zhao, L., Luo, Y., Bhuyan, L. N., and Iyer, R., "A network processor-based Content-aware switch," in *Micro*, 2006.

[65] Zorn, B., "Thoughts on the future of runtime systems," in *IBM Future of VEEs Workshop*, Sept. 2004.